NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

OPNET IMPLEMENTATION OF SPREAD SPECTRUM NETWORK FOR VOICE AND DATA DISTRIBUTION

by

Roger D. Standfield

December, 1997

Thesis Advisor: Murali Tummala
Second Reader: Tri Ha

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1997.	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE
OPNET IMPLEMENTATION OF SPREAD SPECTRUM NETWORK FOR VOICE AND DATA DISTRIBUTION

5. FUNDING NUMBERS

6. AUTHOR(S) Standfield, Roger D.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13.
This thesis presents an OPNET model and simulation of a single cell wireless communications system within a proposed expeditionary warfare communications network. The focus of this thesis is to model and implement data and voice traffic generation, slotted ALOHA medium access control protocol, and direct sequence spread spectrum code division multiple access (CDMA) mechanisms in OPNET. The RF channel is modeled as both a Rayleigh fading channel and a non-fading noise limited channel. Simulation results evaluating the induced BER and multiple access implementation are presented.

14. SUBJECT TERMS OPNET, CDMA,Spread Spectrum, slotted ALOHA,medium access control,expeditionary warfare communications.

15. NUMBER OF PAGES 137

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

OPNET IMPLEMENTATION OF SPREAD SPECTRUM NETWORK FOR VOICE AND DATA DISTRIBUTION

Roger D. Standfield
Captain, United States Marine Corps
B.S., East Central University, 1991

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1997

Author: ______________________________
Roger D. Standfield

Approved by: ______________________________
Murali Tummala, Thesis Advisor

____
Tri T. Ha, Second Reader

____
Herschel H. Loomis, Jr., Chairman
Department of Electrical and Computer Engineering

ABSTRACT

This thesis presents an OPNET model and simulation of a single cell wireless communications system within a proposed expeditionary warfare communications network. The focus of this thesis is to model and implement data and voice traffic generation, slotted ALOHA medium access control protocol, and direct sequence spread spectrum code division multiple access (CDMA) mechanisms in OPNET. The RF channel is modeled as both a Rayleigh fading channel and a non-fading noise limited channel. Simulation results evaluating the induced BER and multiple access implementation are presented.

TABLE OF CONTENTS

I. INTRODUCTION 1
A. OBJECTIVES 2
B. ORGANIZATION 2
II. BACKGROUND 3
A. TRAFFIC MODELING 4
B. MEDIUM ACCESS CONTROL 4
C. SPREAD SPECTRUM 6
D. MODELING THE CHANNEL 11
E. SUMMARY 12
III. MODELING THE TRANSMITTER 15
A. TRAFFIC GENERATION 15
B. MEDIUM ACCESS CONTROL 18
C. SPREAD SPECTRUM 21
D. THE RADIO TRANSMITTER MODULE 23
E. THE TRANSMITTER ANTENNA MODULE 24
F. SUMMARY 25
IV. MODELING THE CHANNEL 27
A TRANSMITTER STAGES 27
B. RECEIVER STAGES 29
C. SUMMARY 33
V. MODELING THE RECEIVER 35
A. THE RECEIVER ANTENNA MODULE 35
B. THE RADIO RECEIVER MODULE 35
C. CONFIGURING THE RECEIVER 36
D. SYNC STATE 39
E. DECODE STATE 41
F. SUMMARY 43
VI. SIMULATION RESULTS 45
A. SIMULATION SETUP 45

B. BIT ERROR RATE EVALUATION 48
C. CODE DIVISION MULTIPLE ACCESS EVALUATION 50
D. EVALUATION OF MAC PROTOCOL PERFORMANCE 51
E. SUMMARY 54
VII. CONCLUSIONS AND RECOMMENDATIONS 55
A. CONCLUSIONS 55
B. RECOMMENDATIONS FOR IMPROVEMENTS 56
APPENDIX A. AN OVERVIEW OF OPNET 57
APPENDIX B. PIPELINE SPECIFICS 67
APPENDIX C. CONSTANT TRAFFIC GENERATOR SOURCE CODE 75
APPENDIX D. SLOTTED ALOHA SOURCE CODE 83
APPENDIX E. PN DESPREADER SOURCE CODE 91
APPENDIX F. PN DESPREADER SOURCE CODE 101
REFERENCES 125
INITIAL DISTRIBUTION LIST 127

I. INTRODUCTION

The direction of both the Navy and Department of Defense has changed over the past few years, with an increased emphasis on expeditionary warfare and littoral operations [1,2,3]. Air-to-ground and ship-to-shore communications are a necessity for littoral operations. A conceptual network which could provide the necessary communications backbone for littoral operations is proposed as portrayed in Figure 1 [4]. The conceived network supports this requirement with a series of routers interconnecting the sea-based forces, aerial elements, and the mobile warfighters ashore.

In an effort to determine the feasibility of this network, this thesis develops a model and simulation of the communications link between the ground based warfighter and an interconnecting router. The link is modeled as a single cell within the network.

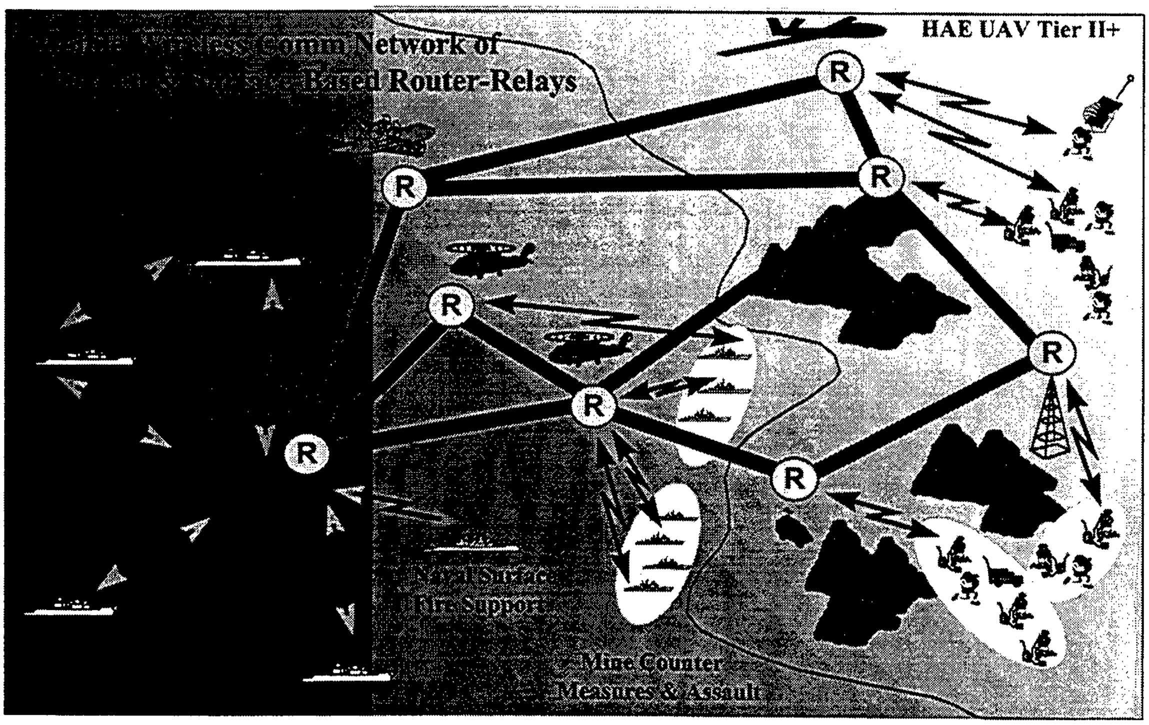

Figure 1. Proposed Expeditionary Warfare Communications Network [from [4]]

A. OBJECTIVES

The objective of this thesis is to develop an OPNET model and simulation of the communications uplink for the proposed system shown in Figure 1. The modeling tool used is MIL 3 Inc.'s Optimized Network Engineering Tool (OPNET), version 3.0.A. This effort concentrates on the medium access control (MAC) layer and the physical layer. The model incorporates data and voice traffic sources. Medium access control is implemented with the slotted ALOHA protocol. The physical layer is implemented as a spread spectrum code division multiple access (CDMA) system. The system is simulated in OPNET and the bit error rate and channel utilization performance results are presented.

B. ORGANIZATION

This thesis is organized as follows. Chapter II provides the theoretical background for the model. Chapter III addresses the implementation of the transmitter. The system traffic models are reviewed, and the details of the slotted ALOHA MAC and direct sequence spread spectrum implementation are covered. Chapter IV covers the characterization of the RF channel using OPNET's pipeline stages. Chapter V outlines the implementation of the receiver and gives the specifics of multiple access reception. Chapter VI presents the simulation results and evaluates the system performance. Chapter VII concludes this effort and offers recommendations for future development.

Appendix A gives an overview of the OPNET modeling software for the reader who is unfamiliar with the package. Appendix B details the specifics of the OPNET pipeline stages. The details of the Interference, SNR, Bit Error Rate (BER), and the Error Allocation pipeline stages are explored. Appendices C through E provide the OPNET model source code for implementing the transmitter, the pipeline stages, and the receiver, respectively.

II. BACKGROUND

This thesis concerns the modeling and implementation of the cellular environment depicted in Figure 2.1 The system is considered a single cell with K nodes. The nodes are assumed static within a cell radius of five kilometers. Power control is implied in that each node is equidistant from the receiver and all transmit equal power. The model includes voice and data traffic generation and focuses on three distinct areas: medium access control, spread spectrum code division multiple access (CDMA), and the radio channel. Figure 2.2 depicts the link between a single node and the receiver. This thesis models each functional block, and this chapter presents the theoretical basis for each block.

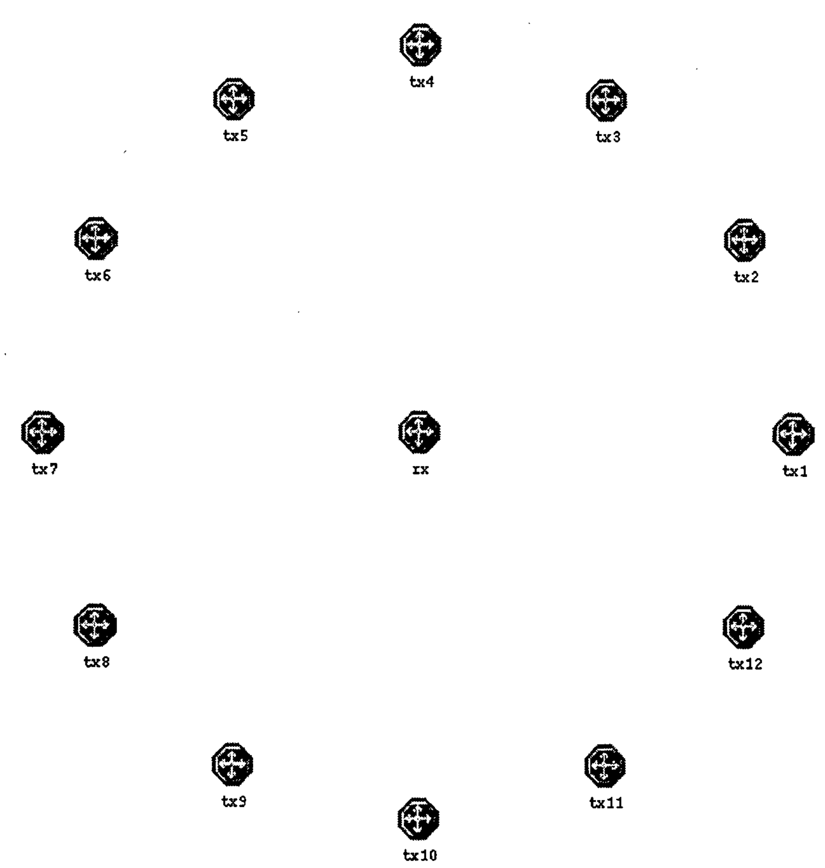

Figure 2.1. Cellular System Overview: Base Station Encircled by K Nodes

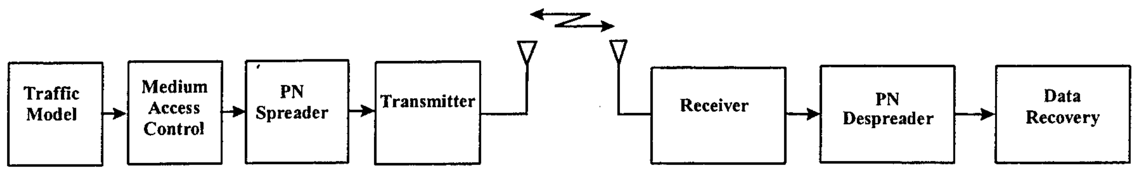

Figure 2.2. Single Communications Link between Node and Receiver

A. TRAFFIC MODELING

The traffic considered for this system is a combination of voice and data packets generated from a user application layer. Considerable effort has been devoted to modeling both voice and data traffic [5]. Here we utilize the traffic models developed by Uziel [6].

Uziel proposes low rate models for voice, video and data. The low rate models are especially applicable in this environment because of the limited channel rate. The voice model is based on Brady's [7] Markovian model for a two-way conversation. The model in [6] is represented as a three state birth-death Markov process. It generates voice traffic at a mean rate of 75 packets/sec.

The data model [6] is represented as a hybrid three state Markov chain consisting of *inactivity*, *low activity*, and *high activity* states. The *low activity* state considers transfer of text files, email messages and such, and this activity is expected to occur frequently. The *high activity* state considers much larger data transfers such as imagery and graphics but is expected to occur much less frequently. Though both low and high active states offer the same data rates, the *low activity* state produces an average of 50 packets/burst while the *high activity* state produces an average of 1,000 packets/burst. Consequently, the model generates data traffic at an average rate of 3.85 packets/sec [6].

B. MEDIUM ACCESS CONTROL

In this section we analyze the theoretical performance of slotted ALOHA medium access control. The performance of access contention techniques can be evaluated by the measuring the system throughput, which is defined as the average number of successful receptions per unit of time. Assuming packet transmissions have a Poisson distribution, the system throughput is defined as $R = \lambda\tau$, where λ is the mean arrival rate and τ is the

packet duration in seconds. The normalized throughput T is defined as the total offered load to the system times the probability of a successful transmission: [8]

$$T = R \times Pr[\text{no collisions}] = \lambda\tau \times Pr[\text{no collisions}].$$

The probability that n packets are generated by the user population during a given packet duration interval, assuming a Poisson distribution, is given as

$$Pr[n] = \frac{R^n e^{-R}}{n!}. \tag{2.1}$$

The probability that zero packets are generated, thus there are no collisions, during this time interval is given by

$$Pr[0] = e^{-R}. \tag{2.2}$$

The time interval in which packets are susceptible to collisions from other packets is referred to as the vulnerable period, V_p [8]. Figure 2.3 depicts the arrival of packets and the vulnerable period $V_p = 2\tau$ where packet collision is possible.

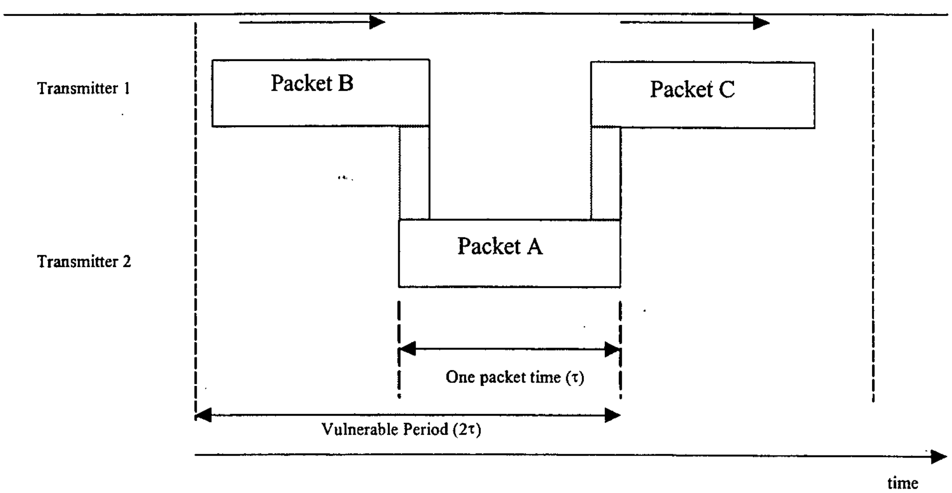

Figure 2.3. Vulnerable period of packet collision [from [8]]

1. Slotted ALOHA

Slotted ALOHA divides the time axis into slots of length greater than or equal to the packet duration τ. All transmitters are synchronized and allowed transmissions only at the beginning of a slot. Although multiple packets may arrive during any slot, transmitters will only transmit one packet at the beginning of each slot. This implementation prevents partial collisions as a packet experiences either total collisions (all packets are the same length) or no collision. [9]

The vulnerable period for slotted ALOHA is τ or one packet duration. The probability that no other packets will be generated during the vulnerable period is e^{-R}. Normalized throughput for slotted ALOHA is obtained as [8,9]

$$T = Re^{-R}. \tag{2.3}$$

The maximum throughput is found by taking the derivative of Equation (2.3), and solving for *R:*

$$\frac{dT}{dR}\left(T = \mathrm{Re}^{-R}\right) = e^{-R} - \mathrm{Re}^{-R} = 0 \tag{2.4}$$

which yields $R_{\max} = 1$. The maximum throughput of the slotted ALOHA system can be found by substituting $R_{\max} = 1$ into Equation (2.3):

$$T = e^{-1} = 0.3679.$$

C. SPREAD SPECTRUM

In this section we review the spread spectrum concepts of code division multiple access (CDMA) techniques. Direct sequence (DS) and frequency hopping are two common techniques. Here we consider DS CDMA only. For a detailed study of spread spectrum techniques, the reader is referred to [10].

1. Direct Sequence Spread Spectrum

Implementing DS spread spectrum requires the generation of a pseudo-random signal (PN sequence) resembling a binary wave form, followed by the spreading of the information signal with the generated sequence.

a. Generating the PN Sequence

PN sequences (codes) are typically generated using sequential logic circuits consisting of a shift register of n stages (*degree n*) and a feedback loop. We focus on linear feedback shift-registers as depicted in Figure 2.4. The shift register is initialized with an initial binary load. With each clock pulse, the binary sequences are shifted through the register, and the output of selected stages of the register are tapped and fed back to the feedback logic. The feedback logic consists of an exclusive-OR gate, whose output becomes the input to the register.

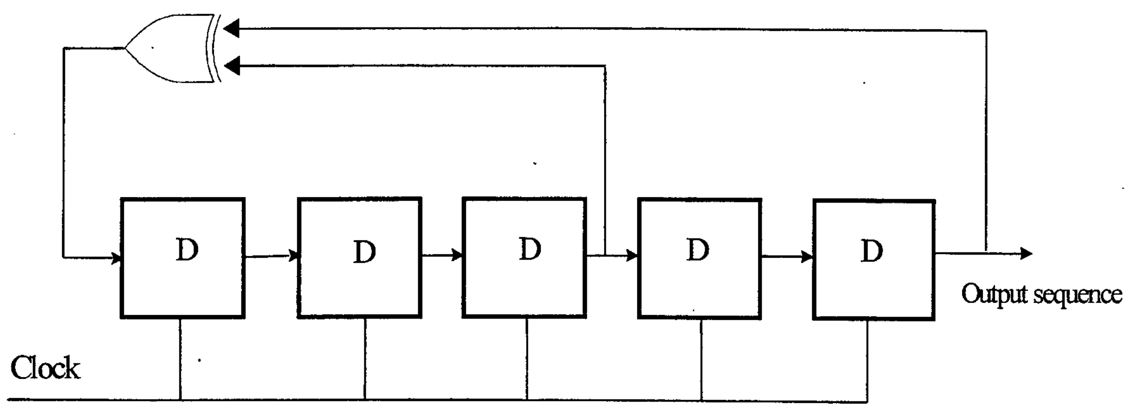

Figure 2.4. Five Stage Linear Feedback Shift Register for Generating PN Sequences

If the initial condition (IC) of the shift register in Figure 2.4 is set with a binary sequence that causes the register to cycle through all possible (2^n) states, the PN sequence generated is known as a *maximal length sequence* (ML) or *m*-sequence. The all zeros state is not allowed. Thus, an *m*-sequence shift register produces $2^n - 1$ states.

The feedback taps of the shift register can be described as the characteristic polynomial. The polynomial characterizing the feedback in Figure 2.4 is $1 + D^3 + D^5$.

This characteristic polynomial can be denoted $R(x_i)$, $i = 1,2,..,n$, where x_i indicates the feedback stages. For example, $R(3,5)$ fully describes the register of Figure 2.4. [10,11].

Not all polynomials produce *m*-sequences. Those that do are known as *primitive polynomials*. The number of primitive polynomials of degree n is found from [10]

$$N_p = \frac{2^n - 1}{n} \prod_{i=1}^{J} \frac{p_i - 1}{p_i} \tag{2.5}$$

where J is the number of prime factors and p_i are the prime factors of $2^n - 1$. As an example, consider the shift register of Figure 2.4. The only prime factor of $2^5 - 1$ is 31. Thus, $i = J = 1$, $p_1 = 31$, and $N_p = 6$.

b. Spreading the Signal

Spreading the information signal is accomplished by the multiplication of the information signal with the PN sequence as depicted in Figure 2.5. The message sequence and the PN sequence must be transformed from normal binary sequences to sequences of +1s and −1s. The resultant signal is modulated with a carrier and becomes the actual transmitted signal.

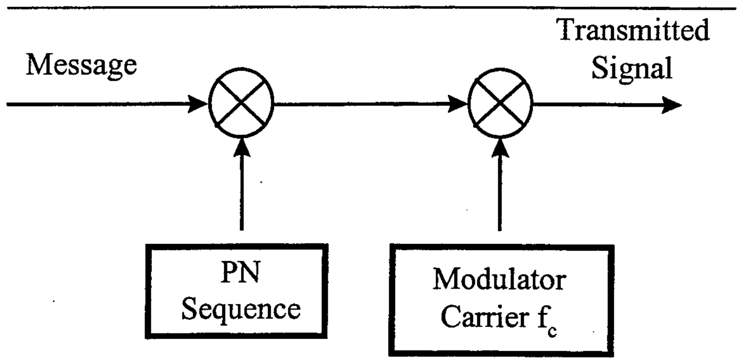

Figure 2.5. Simplified DS-SS Transmitter [from [8]]

Figure 2.6 depicts an information signal waveform and the corresponding PN sequence waveform generated using the configuration of Figure 2.4 with an initial load of 00101. The PN sequence is multiplied with the information signal. The resulting signal is equivalent to the PN sequence for all data bit 1s and the inverse of the PN sequence for

all data bit 0s. T_b is the time duration of the data bit pulse, and T_c is the time duration of the PN code pulse, often referred to as *chip duration*.

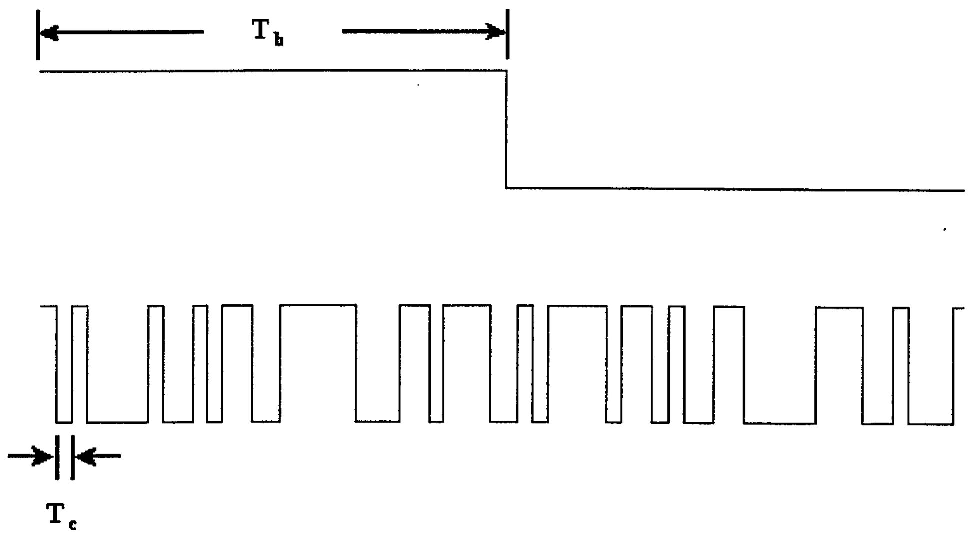

Figure 2.6. Information and PN Sequence Waveform

As seen in Figure 2.6, this implementation produces 31 chips per information bit. The information bit rate $R_b = 1/T_b$ and the chip rate $R_c = 1/T_c$. Defining $k = 2^n - 1$, the number of chips per bit, we have

$$T_b = kT_c \tag{2.6}$$

or equivalently

$$R_c = kR_b \,. \tag{2.7}$$

The information data bits are spread with the complete PN sequence, such that the PN sequence repeats completely every bit interval. Such an implementation is referred to as one based on *short* codes. Spreading of the information bits results in what is termed as the *processing gain* of the system and can be shown as [Sklar]

$$G_{DS} = \frac{B_{SS}}{B} \tag{2.8}$$

where B_{SS} is the spread spectrum bandwidth and B is the minimum bandwidth of the data, normally taken to be the data rate R_b [10,12]. For short codes, *processing gain* can be equivalently expressed as [11]

$$G_{DS}\text{ (dB)} = 10\log_{10}(k)\,. \tag{2.9}$$

c. *Despreading the Signal*

The receiver's objective is to recover the spreaded information in the received signal into its original form. In addition to the desired signal, the received signal consists of possible jamming interference and noise [10]. Figure 2.7 offers a functional block diagram of a Direct Sequence Spread Spectrum receiver. Assuming synchronization at the receiver, the received signal is multiplied by a locally generated replica of the PN sequence and demodulated with the local carrier. The resulting signal is

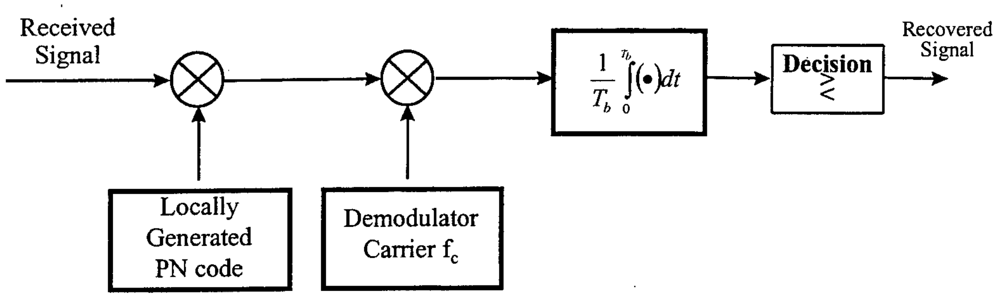

Figure 2.7. Simplified DS-SS Receiver: $K = 1$ User [from [8]]

integrated over a one-bit duration T_b followed by a decision block. The receiver repeats this multiplication of the received signal with each known PN sequence. Thus, it can differentiate N_P different signals. The resulting receiver can be envisioned as a bank of correlators. Figure 2.8 depicts N_P crosscorrelations of the received signal.

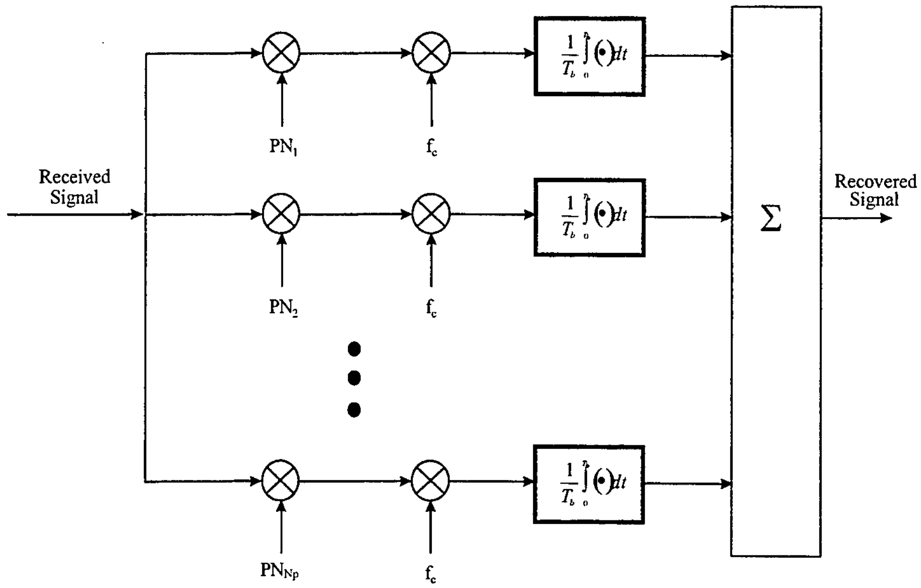

Figure 2.8. Simplified DS-SS Receiver: Bank of Correlators [from [8]]

D. MODELING THE CHANNEL

Multiple paths, signal fading, shadowing, reflections, and propagation loss are unique concerns in the radio environment [8,13]. Path losses in radio channels have been studied extensively and several models are proposed in the literature [8]. This thesis implements the 2-ray model as defined in [8]. The defined received power is

$$P_r = P_t G_t G_r \frac{h_t^2 h_r^2}{d^4} \tag{2.10}$$

where P_t is the transmitted power, G_t and G_r are the respective transmitter and receiver antenna gains, h_t is the transmitter antenna height, h_r is the receiver antenna height, and d is the separating distance (meters) of the transmitter-receiver pair. For large values of d, the received power and path loss become independent of frequency in the 2-ray model [8]. Large distances are considered valid where $d \gg \sqrt{h_t h_r}$. For the system being considered, we have the following typical values: d = 5 km, $h_t \leq 20$ meters, and $h_r \leq 2$ meters, hence the 2-ray model is considered valid for this implementation.

The bit error rate (BER) is one of the most critical components of channel characterization. The BER may be a function of several parameters, depending on the

channel model used. Multipath and fading are major contributors to the radio channel bit error rate. The probability of error in a Rayleigh fading channel (coherent BPSK) is given as [14]

$$P_e = \frac{1}{2}\left(1 - \sqrt{\frac{\overline{\gamma_b}}{1+\overline{\gamma_b}}}\right) \tag{2.11}$$

where

$$\overline{\gamma_b} = \frac{E_b}{N_0}\,\mathrm{E}\left(\alpha^2\right).$$

$\mathrm{E}(\alpha^2)$ is the average value of α^2, the attenuation factor, and

$$\frac{E_b}{N_0} = SNR - 10\log_{10}(R_b). \tag{2.12}$$

The standard Gaussian approximation (SGA) in [15] is often used in determining the bit error probability in CDMA systems [8,16]. For a non-fading, noise-limited channel with perfect power control, the average probability of bit error is given as [8]

$$P_e = Q\left(\sqrt{\frac{1}{\frac{K-1}{3k} + \frac{N_0}{2E_b}}}\right) \tag{2.13}$$

where K is the number of multiple users and k is the number of chips per information bit assuming short code implementation.

This thesis incorporates BER modeling for both a Rayleigh slow fading channel and a noise limited channel without fading. The BER calculations are based on Equations (2.11) and (2.13), respectively.

E. SUMMARY

This chapter reviewed the modeled system as single cell consisting of a base station and K nodes. We briefly reviewed the traffic models that will be employed and medium access control and CDMA techniques as they pertain to this model. The

propagation model and BER models for channel characterization were given. The next three chapters detail the implementation aspects of medium access control, CDMA , and the radio channel using the OPNET modeling software.

III. MODELING THE TRANSMITTER

The goal of this and the following chapters is to detail the OPNET implementation of the proposed system. This chapter addresses the implementation issues of the transmitter. The transmitter can be divided into three functional areas: traffic generation, access control to the radio channel, and the physical interface of the transmission. Figure 3.1 offers a graphical view of the transmitter model, consisting of three user defined processes and the OPNET supplied radio transmitter and antenna modules. The objective of this chapter is to model each functional block.

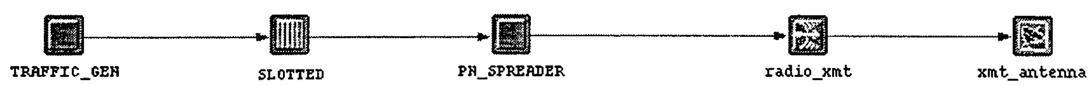

Figure 3.1. Transmitter Node Model

A. TRAFFIC GENERATION

This section reviews the traffic models and their packet formats. Three different traffic models are employed. The voice and data models outlined in [6] were modified as necessary for this effort. A third traffic model generating a constant rate of traffic is added.

1. Upper Layer Packet Format

As this effort is focused on the lower layers of the communications system, arriving packets are considered a single entity. Any required packet segmentation is assumed to have occurred prior to the packet's arrival. The lower layers' addressing and other responsibilities are not implemented.

The packet format for all three traffic models is the same. Regardless of the traffic source, the packet generated is formatted as shown in Figure 3.2.

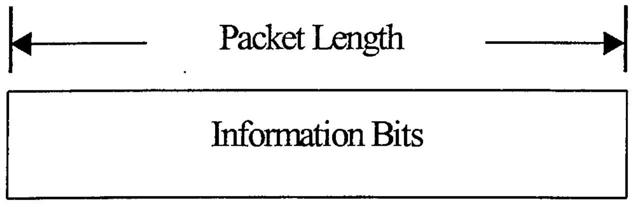

Figure 3.2. Upper Layer Packet Format

2. Data and Voice Traffic

The data traffic generation process consists of the states depicted in Figure 3.3. The model generates traffic at an average rate of 3.85 packets per second for each of N_D data sources, where N_D is user definable [6].

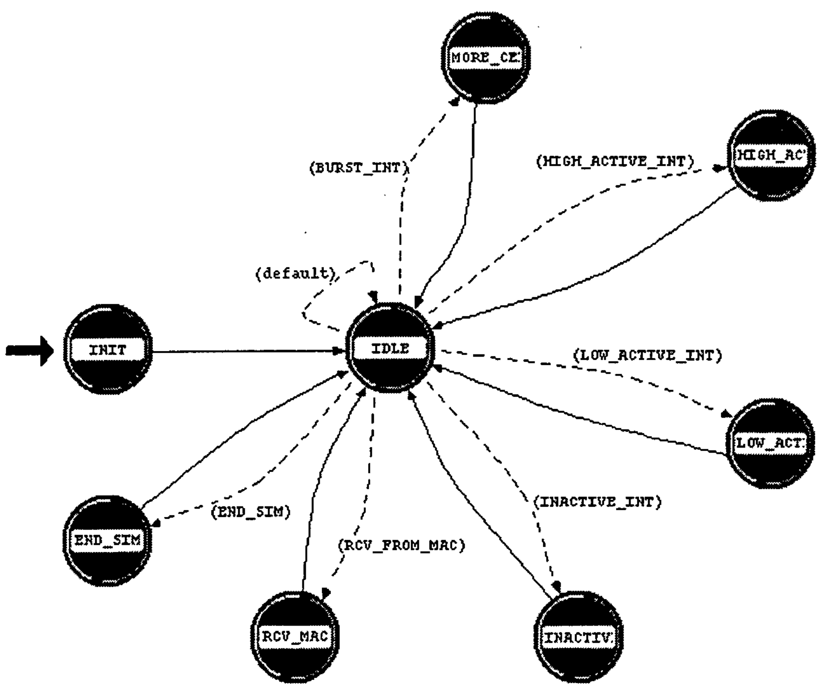

Figure 3.3. State Transition Diagram of Data/Voice Traffic Generation Process[from [6]]

The voice model is graphically quite similar to the data model shown in Figure 3.3, differing in that it models a two way conversation and offers a higher rate of traffic. Whereas the data model generates traffic for N_D sources, the voice model generates traffic for N_S two-way speech conversations at an average rate of 75 packets per second per conversation.

3. Constant Rate Traffic

The constant rate traffic generator depicted in Figure 3.4 is functionally quite similar to OPNET's ideal source generator. It differs in that it generates and transmits packets of the format shown in Figure 3.2 at a constant rate.

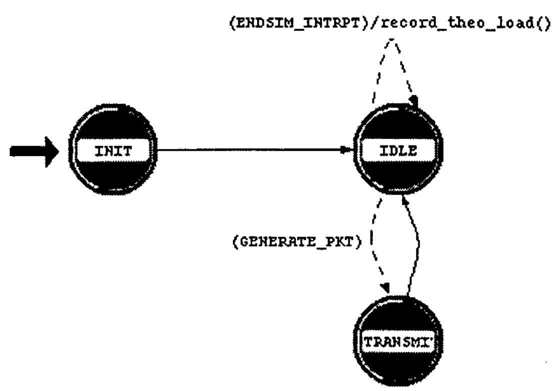

Figure 3.4. State Transition Diagram of Constant Rate Traffic Generation Process

4. Init State

The activity performed in the INIT state (Figures 3.3 and 3.4) is common to all three traffic models. Within the INIT state, the two user definable simulation parameters listed in Table 3.1 are queried for and obtained. The *Active Sources* parameter determines the number N_D of data or N_S of voice sources the model will generate, respectively. For example, with $N_D = 10$, the data model of [6] shown in Figure 3.3 will produce 10 separate data sources, each generating an average of 3.85 packets/sec. *User Data Bits* is

the user's choice for a data bit stream (0001110...). This bit stream is repeated over the length of the packet to set the bits of each packet and represents the information data bits received from the higher layers.

Table 3.1. Traffic Generator Simulation Parameters

Simulation Parameter	Data Format
Active Sources	Integer
User Data Bits	Character String

B. MEDIUM ACCESS CONTROL

The reader might consider medium access control similar to access control of a busy freeway. Consider the RF radio channel medium as the freeway. In simplistic terms, this is the functionality of the MAC layer. In this section, we offer the models' implementation.

1. MAC Layer Packet Format

We first consider the formatting of the packet performed at this level. Figure 3.5 depicts the MAC layer packet format. A MAC header is provided, but the addressing and other information normally present in the header is not implemented. The MAC header for this model consists simply of all 1s.

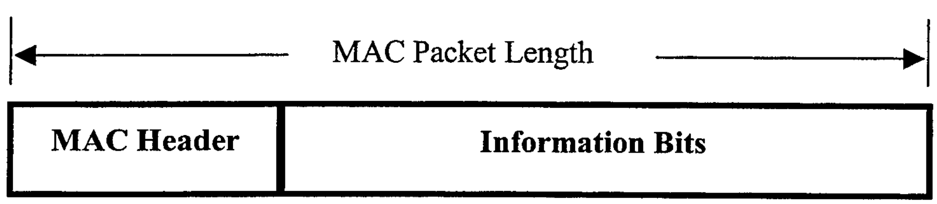

Figure 3.5. MAC Layer Packet Format

2. Slotted ALOHA Implementation

In this section we discuss the INIT, IDLE, QUEUING, and TRANSMIT states of the slotted ALOHA process shown in Figure 3.6. The functions of this process consist of

enqueuing received information packets and transmitting them to the physical layer at the appropriate slot time. Prior to transmitting them, however, the MAC header is added to the information packet to create the MAC layer packet format shown in Figure 3.5.

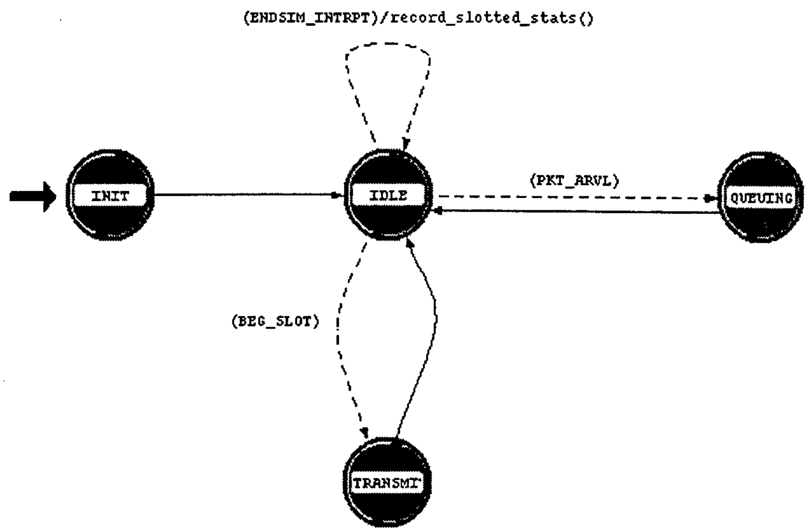

Figure 3.6. State Transition Diagram of the Slotted ALOHA Process

a. Init State

Within the INIT state variables are initialized and the user definable simulation parameters shown in Table 3.2 are queried and obtained. *Slot length* allows the

Table 3.2. Slotted ALOHA Process Simulation Parameters

Simulation Parameter	Data Format
Slot Length	Double
Chip Rate	Double
Guard Band	Integer

user to define a slot length of the their choice. If the user accepts the models' default implementation, we determine slot length (seconds) as

$$T_S = \frac{L_P}{R_c} \tag{3.1}$$

where L_P is the packet length (bits) and R_c is the chip rate of the transmitter (bps for this notation; at this level, bits/chips are synonymous). Note that L_P is not the MAC packet length in bits, rather the bit length of the packet which will be transmitted from the physical layer transmitter. Although this packet is not actually dealt with until the next section, it is important at this point to consider the spread packet that is actually transmitted.

The actual packet format which will reach the physical layer was shown in Figure 3.5. This model implements limited synchronization functionality by prefacing the MAC packet with a physical layer synchronization frame. A notional guard band frame is added as a trailer. Figure 3.7 depicts the notional physical layer packet format. Before actually being transmitted, this packet will be spread by the transmitter's PN code.

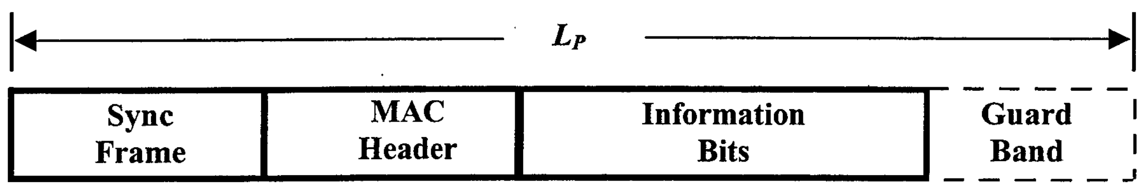

Figure 3.7. MAC View of the Physical Layer Packet Format

This spread packet has length L_P which the MAC process must account for in determining the slot length. Accounting for this spreading then,

$$L_P = (\text{Sync Bits } + \text{MAC Header} + \text{Information} + \text{Guard Band}) \times G_{DS}\,. \tag{3.2}$$

The actions performed within the INIT state are accomplished at the beginning of the simulation, effectively at time t = 0. Upon completion, the process is forced to the IDLE state to await a condition invoking a transition to another state. The choices are the QUEUING or TRANSMIT state.

b. Queuing and Transmit States

Transition to the QUEUING state is invoked upon receiving a packet from the upper layer. In OPNET terms, the simulation kernel (SK) delivers an interrupt

indicating a packet arrival. Within the QUEUING state the received packet is enqueued and the process returns to the IDLE state. At every slot time T_S a transition is made to the TRANSMIT state. If the queue has packets, one is retrieved, a MAC header is added to create the packet format shown in Figure 3.5, and the packet is sent to the physical layer. The process then immediately returns to the IDLE state to await either another packet arrival or another slot time.

C. SPREAD SPECTRUM

1. Physical Layer Packet Format

The task of the physical layer is to transmit the packet received from the MAC layer to the channel. The packet is normally prefaced with physical layer information consisting minimally of a synchronization frame [9]. Chapter V discusses the specifics of OPNET in that the SK provides implicit synchronization. Synchronization in a spread spectrum system is beyond the scope of the work reported here. We note its requirement by including nominal synchronization functionality.

The physical layer packet format shown in Figure 3.7 has a notional guard band attached. The packet format without the guardband is shown in Figure 3.8.

Sync Frame	MAC Header	Information Bits

Figure 3.8. Physical Layer Packet Format

The synchronization frame is defined similar to the MAC header, consisting of all 1s. Prior to transmission, the complete packet is spread with the PN sequence assigned to this transmitter.

2. Spread Spectrum Implementation

In this section we discuss the process of implementing spread spectrum. The process consists of the INIT, IDLE, and SPREADING states shown in Figure 3.9.

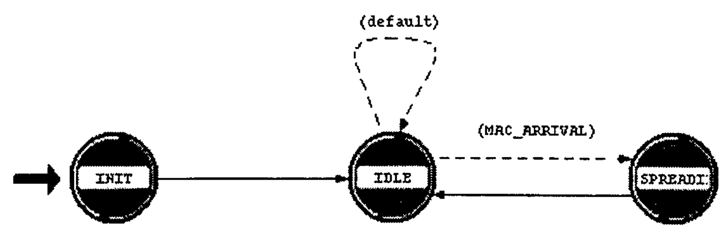

Figure 3.9. State Transition Diagram of the Spreader Process

a. INIT State

Within the INIT state variables are initialized and the user defined simulation parameters listed in Table 3.3 are queried and obtained. *User Register Load* defines the initial condition of the shift register used in generating the transmitter's PN sequence. *Polynomial* is the characteristic polynomial defining the feedback taps of the shift register as discussed in section 2.C. Implied is the static allocation of the transmitter's PN code. It is determined at initialization based on *Polynomial.*

Table 3.3. Spreader Process Simulation Parameters

Simulation Parameter	Data Format
User Register Load	Character String
Polynomial	Character String

Dynamic assignment or allocation of PN codes is not implemented. The PN code generated for the transmitter is determined within this state and maintained for persistence. It is used repeatedly by the TRANSMIT state for the actual spreading of the packet.

b. IDLE State

Upon completing initialization, the process is forced into the IDLE state where it awaits traffic from the upper MAC layer. Transition from IDLE is conditioned on MAC_ARRIVAL, defined as receiving a packet from the MAC layer. This event

indicates the slotted ALOHA process has reached a slot time and has transmitted an enqueued packet to the physical layer for transmission.

c. *Spreading State*

This state implements the physical transmission of the MAC packets. After obtaining the incoming packet, synchronization bits are prefaced on the packet and the combined packet is spread with the PN code.

At this point the packet consists of a sequential string of 1s and 0s without discernible boundaries. This bitstream is encoded as a sequence of 1s and –1s. Figure 3.8 offers a representative view. The actual mapping is irrelevant; it is only necessary that the receiver is consistent and knowledgeable of the encoding scheme. As defined in Chapter II, an information bit 1 is spread with the PN sequence. An information bit 0 is spread with the inverse of the PN sequence. The spread 0s are encoded as –1s and the spread 1s are encoded as +1s.

D. THE RADIO TRANSMITTER MODULE

OPNET provides a built-in radio transmitter module, requiring only that the user properly define the values for the attributes listed in Table 3.4.

Table 3.4. Radio Transmitter Attributes

ATTRIBUTE	VALUE
Name	User's Choice
Channel	(...)
Modulation	BPSK, QPSK, DPSK, ...
RX Group Model	User's RX Group Model
TX Delay Model	User's TX Delay Model
Closure Model	User's Closure Model
Channel Match Model	User's Channel Match Model
TX Antenna Gain Model	Isotropic or User's model
Propagation Delay Model	User's Propagation Delay Model
Icon Name	User's Choice

The values of the various *Model* attributes are user defined files known as pipeline stages. The requirements are the proper naming of the C source code file that implements the respective pipeline stage. The pipeline stages are the implementation of the RF channel, and in effect, provide the link analysis. Chapter IV is devoted to channel characterization and provides the details of each stage.

Modulation is used internally by OPNET to determine the bit error rate. It is based on OPNET's internally defined bit error rate tables. We will see in Chapter IV where we provide this model's implementation of bit errors. Thus, this parameter is not applicable to this model.

The *channel* attribute listed in Table 3.4 is further characterized by the parameters listed in Table 3.5. C_i represents the respective channel of which there may be several.

Table 3.5. Transmitter Channel Parameters

C_i	*Data Rate*	*Packet Format*	*Bandwidth (kHz)*	*Minimum Frequency (MHz)*	*Spreading Code*	*Transmitted Power*

The *data rate* parameter is typically the information data rate. This parameter is queried in the pipeline stages and used, for instance, in the transmit delay stage to determine the time delay for transmitting the packets. It is in reality the chip rate.

The *packet format* parameter defines the particular packet format utilized on this channel. This parameter can be all inclusive to accept packets of any format and/or nonformatted packets, or can selectively allow access to the channel for only defined formats. The *bandwidth*, *minimum frequency*, *spreading code*, and *transmitted power* are self explanatory user definable parameters. The *spreading code* parameter is of limited use in this model. This parameter accepts either a Boolean value or an integer. It does not provide the functionality needed for evaluating PN codes on a bit basis. It is disabled in all transmitter channels.

E. THE TRANSMITTER ANTENNA MODULE

OPNET also provides the antenna module. The module permits modeling of directional or isotropic antennas. Interested reader may refer to the OPNET manuals [17]

for a detailed discussion on modeling directional antennas. This model employs isotropic transmitting antennas.

F. SUMMARY

This chapter detailed the implementation of the transmitter. We have followed the packet flow from generation through the physical spreading. The original information packet now has an additional MAC header and synchronization frame, and this complete packet has been spread with the transmitter's PN code. The packet is now destined for a receiver, but it will first need to traverse the RF channel. The packet undergoes modifications and suffers loss in the form of bit errors as it traverses the channel. The next chapter details the model's channel characterization.

IV. MODELING THE CHANNEL

Channel characterization is a key component of any radio communications model and there are several critical issues to evaluate. This chapter addresses the implementation of these issues. Modeling radio channels within OPNET requires the 14 pipeline stages shown in Figure 4.1. The objective of this chapter is to model the RF channel utilizing these stages. This chapter presents the implementation of each stage with particular focus on the Interference, SNR, BER, and Error Allocation stages.

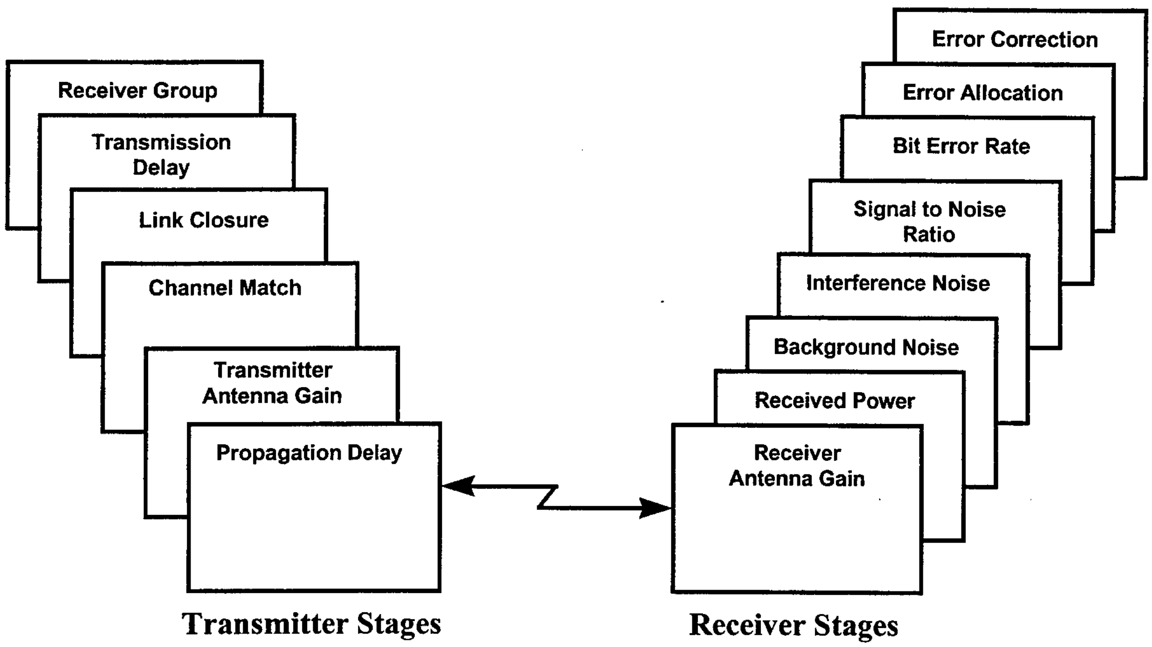

Figure 4.1. OPNET's Radio Channel Pipeline Stages

A. TRANSMITTER STAGES

1. Receiver Group

The receiver group stage is not part of the dynamically executed stages of the remainder of the pipeline. It is executed only once at the beginning of the simulation. As radio transmissions are initiated, OPNET's simulation kernel (SK) establishes a destination channel set between all known transmitters and receivers in the system. For each receiver the transmitter maintains a separate channel set; together all the channel sets

make up a destination list. During transmission (simulation), the pipeline stages will be executed for each channel set in this destination list. In this receiver group stage, the designer has the opportunity to exclude a receiver from a transmitter's destination list. By default, OPNET does not exclude any receivers in the network and allows the pipeline stages to dynamically make the determination. We employ the default for the receiver group stage.

2. Transmission Delay

This stage determines the time delay for transmitting the packet. The delay is computed as the packet bit length, L_P, divided by the channel data rate, R_c:

$$tx_{delay} = \frac{L_P}{R_c} . \tag{4.1}$$

Note that R_c is the channel attribute *data rate* discussed in section 3.D. The delay value obtained is placed in the appropriate Transmission Data Attribute (TDA) of the respective packet.

3. Link Closure

The link closure stage determines if a particular receiver can be affected by a transmission. The functionality of this stage is similar to the receiver group stage, but it is executed dynamically with every packet. Link closure is determined based on the line-of-sight propagation path assumption between the transmitter and the receiver. By default, OPNET utilizes a ray-tracing line-of-sight algorithm with the earth's surface modeled as a sphere [17]. The Boolean value indicating link closure is placed in the appropriate packet TDA. This model implements the OPNET default pipeline stage.

4. Channel Match

The channel match stage classifies transmitted packets as one of three categories; **valid**, **noise**, or **ignored**, with respect to a particular receiver. **Ignored** packets are not further evaluated by the remaining stages.

Channel match is a function of *frequency*, *bandwidth*, *data rate*, *spreading code*, and *modulation type* of both the transmitter and receiver. These are user definable parameters discussed in section 3.D. Packets with non-overlapping *frequencies* between

the transmitter-receiver (tx-rx) pair are classified as **ignored**. Packets with one or more mismatched parameters are classified as **noise**. Packets whose set of tx-rx parameters are completely matched are **valid**.

As the *spreading code* attribute is disabled in all transmitters and receivers, this assures no link will be classified based on the PN code within this stage. Chapter V will discuss how we determine a PN code match or mismatch in the receiver.

5. Transmitter Antenna Gain

This stage determines the gain of the transmitting antenna. The proposed system is modeled using isotropic transmitter and receiver antennas, which is simplistic in terms of necessary calculations. The uniform gain (0 dB) of the transmitting antenna is set in the appropriate packet TDA during the execution of this stage.

6. Propagation Delay

The propagation delay stage is the final pipeline stage associated with the transmitter. The propagation delay determines the time lapse between the transmission of the first bit of a packet at the transmitter and the arrival of the first bit at the receiver. As either transmitter or receiver may be mobile, two calculations are performed. The first calculation accounts for the distance separating the tx-rx pair at the beginning of packet transmission, and the second for the distance separating the tx-rx pair at the end of packet transmission. The propagation delays are calculated as

$$t_{prop} = \frac{d_i}{C} \tag{4.2}$$

where d_i is the respective start/end transmission distance, and C is the velocity of radio wave propagation.

B. RECEIVER STAGES

The remaining stages are associated with the receiver. In these stages the model differs from OPNET's default pipeline stages. A thorough understanding of the detailed workings of the OPNET software is critical to correctly model this system. Of critical importance are the Interference Noise, the SNR, and the BER stages. Appendix B

provides a detailed discussion of the model's implementation of these three pipeline stages.

1. Receiver Antenna Gain

This stage follows in concert with the transmitter. Similar to the transmitting antennas, the receiver antenna is modeled as isotropic. A uniform gain (0 dB) is set in the appropriate TDA.

2. Receiver Power

This stage determines the received power level of the incoming packet. It is calculated in accordance with Equation (2.9) and set in the packet's TDA.

3. Background Noise

The background noise stage models noise sources other than interference noise sources introduced by other transmitters or other intentional interfering sources. The major noise contribution is due to the thermal noise power:

$$N_B = kT_S B \quad \text{(Watts)} \qquad (4.3)$$

where k is Boltzmann's constant, $T_S = T_R + T_0$, and B is the effective *bandwidth* of the receiver. *Bandwidth* is the user definable channel attribute (see section 3.D). The effective receiver temperature T_R is determined as

$$T_R = (F-1)T_0 \qquad \text{°K} \qquad (4.4)$$

where F is the receiver *noise figure* and T_0 is the background source temperature, taken to be 290 °K. The receiver *noise figure* is a user definable attribute. The default model includes a negligible contribution of ambient noise. As this ambient noise is negligible, it is neglected in this model. The determined noise contribution N_B is set in the appropriate packet TDA.

4. Interference Noise

This stage accounts for noise interference that packets may impose upon each other. In effect, it accounts for multiple access interference (MAI). This stage is invoked from the SK only in the case of overlapping packet reception and only at the beginning of packet reception.

The interference power, N_i , on the respective packet segment is determined by

$$N_i = \sum_{i=1}^{M} P_i \tag{4.5}$$

where M is the instantaneous number of interfering packets (multiple users), and P_i is the received power of the respective packet. In this implementation, all transmitters transmit equal power and are equidistant from the receiver, thus $N_i = MP_r$, where P_r is given in Equation (2.9).

5. Signal-to-Noise Ratio

This stage determines the SNR on a packet segment based on the background noise during the segment interval. The interfering and background noise are determined in the previous two stages. The background noise is constant over the length of the packet. In the case of packets of different lengths, N_i may vary over the length of the packet with a corresponding variation in the SNR. By default, OPNET determines the SNR as

$$SNR\,(\text{dB}) = 10\log_{10}\left(\frac{P_r}{N_i + N_B}\right) \tag{4.6}$$

where P_r is the received power, N_i and N_B are the interference and the background noise, respectively.

In the Bit Error stage, the multi-user interference is accounted for in the bit error calculation. The SNR pipeline stage is thus modified and the resultant SNR is only due to the signal to thermal noise ratio, defined as

$$SNR\,(\text{dB}) = 10\log_{10}\left(\frac{P_r}{N_B}\right). \tag{4.7}$$

6. Bit Error Rate

This stage determines the packet segment BER. For packets of equal length, the BER is constant over the length of the packet. Otherwise, it will vary in accordance with the SNR. Here we implement two BER models. For modeling fading channels, the BER is determined by Equation (2.12). Non-fading channels are modeled by Equation (2.14). In this stage, we account for the interfering users whose contribution was omitted in the SNR stage. The calculated BER is set in the packet's TDA.

7. Error Allocation

This stage is executed immediately after the BER pipeline stage completes execution. In this stage, we inject errors in the packet at the bit error rate determined by the BER stage. This stage determines the packet segment length defined in the BER stage, and sequences over this segment of the packet randomly inverting bits within the packet at a rate equal to the BER. As the bits are encoded as +1s and –1s, selected +1s are set to –1, selected –1s become +1.

This model employs packets of equal length. Thus, the packet segment length is equal to the packet length, L_P. The approach to injecting errors can be shown with the following pseudocode.

```
for(i = 0; i < L_P ; i ++)
{
  random_number = rand(0,1);
  if(random_number ≤ BER)
    invert_packet_bit(i);
}
```

where **invert_packet_bit()** indicates changing the sign of the respective bit as noted in the previous paragraph.

8. Error Correction

This stage determines the acceptability of the packet. This determination is based upon two criteria: the availability of the receiver and the bit error rate of the packet.

Should the receiver become disabled during packet reception, the respective packet is rejected and the SK automatically destroys the packet. If the rate of actual errors injected in the Error Allocation stage exceeds the *error correcting threshold*, the packet is rejected; again, the SK automatically destroys the packet. The *error correcting threshold* is a receiver channel attribute to be discussed in Chapter V.

C. SUMMARY

This chapter encompasses the second of the three modules of this model. Presented was the implementation of the RF channel characterization. The first section reviewed the pipeline stages associated with the transmitter. The second reviewed the stages associated with the receiver, where we reviewed the model's implementation of determining the BER and injecting the bit errors accordingly. The original information has thus undergone several changes: a MAC header and a synchronization frame are added, the bits have been spread, and errors have been injected into the packet. The final implementation aspect of this model is to determine if the packet is recoverable and the original information retrievable. Chapter V details the receiver implementation.

V. MODELING THE RECEIVER

This chapter presents the receiver implementation of the model. The objectives of this chapter are to properly model limited synchronization, code division multiple access, and signal despreading in a spread spectrum system. Figure 5.1 offers a graphical view of the receiver model. This chapter focuses on the functions of the despreader and details the OPNET implementation of this spread spectrum receiver.

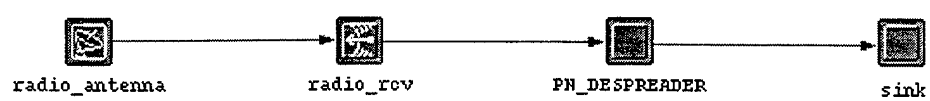

Figure 5.1. OPNET Receiver Node

A. THE RECEIVER ANTENNA MODULE

Similar to the transmitter, the receiver employs the OPNET isotropic antenna module.

B. THE RADIO RECEIVER MODULE

OPNET provides a built-in radio receiver module, requiring only that the user properly define the values for the attributes listed in Table 5.1. The values for the various *Model* attributes are the user defined receiver pipeline stages discussed in Chapter IV. The requirements are the naming of the C source code files which implements the respective pipeline stage.

The value of *Noise Figure* is queried for in the Background Noise pipeline stage and is necessary to determine the receiver temperature, T_R, in Equation (4.4). *ECC Threshold* is the percentage of errors per packet that the receiver will accept; it is queried for in the Error Correction pipeline stage.

Table 5.1. Receiver Module Attributes

ATTRIBUTE	VALUE
Name	User's Choice
Channel	(…)
Modulation	BPSK,QPSK,DPSK, …
Noise Figure	User Defined
ECC Threshold	User Defined
RX Antenna Gain Model	Isotropic or User's model
Power Model	User's Power Model
Background Noise Model	User's BNoise Model
Interference Noise Model	User's INoise Model
SNR Model	User's SNR Model
BER Model	User's BER Model
Error Model	User's Error Model
ECC Model	User's ECC Model
Icon Name	User's Choice

The receiver *Channel* attribute is further characterized by the parameters listed in Table 5.2. These parameters are identical to those in the transmitter with the exception of *processing gain* (see section 2.C.1.b).

Table 5.2. Receiver Channel Parameters

C_i	*Data Rate*	*Packet Format*	*Bandwidth (kHz)*	*Minimum Frequency (MHz)*	*Spreading Code*	***Processing Gain***

C. CONFIGURING THE RECEIVER

This section reviews the configuration of the modeled receiver. A spread spectrum receiver is capable of receiving multiple packets simultaneously. To successfully receive

multiple packets, the receiver must first be able to synchronize with the arriving packet(s). This model implements limited synchronization, which we define as successfully decoding the synchronization frame of the arriving packets. In order to decode the synchronization frame, the receiver msut first despread the arriving packet's synchronization frame. Despreading is accomplished by crosscorrelating the arriving packet's synchronization frame with PN codes generated within the receiver. The receiver needs multiple PN codes with which it can attempt to despread the arriving packet. In configuring the receiver, we generate these multiple PN codes initially.

The complete despreading process consists of the INIT, IDLE, SYNC and DECODE states shown in Figure 5.2. A detailed discussion of each state is presented below.

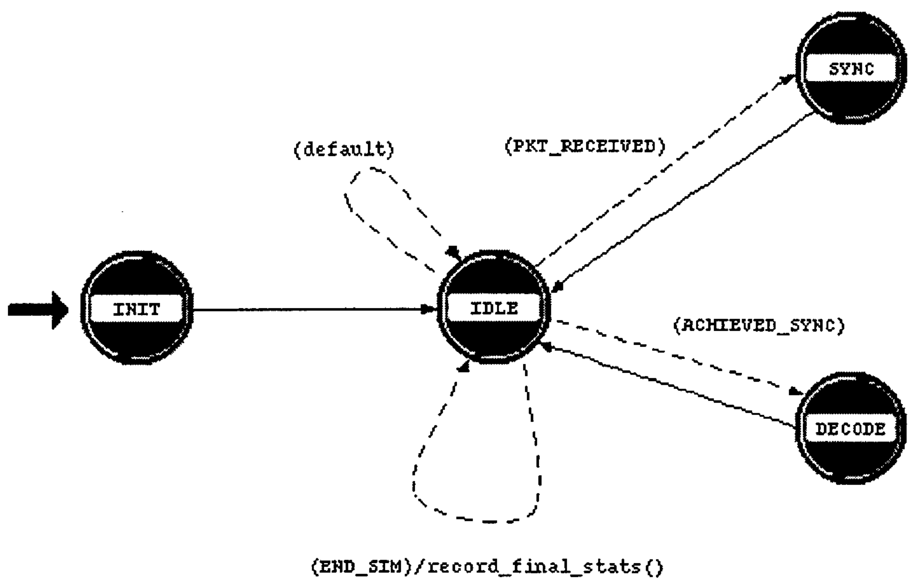

Figure 5.2. State Transition Diagram of the Despreader Process

1. INIT State

Within the INIT state, variables are initialized and the simulation parameters shown in Table 5.3 are queried and obtained. *Synchronization threshold* and *data threshold* are thresholds which must be reached to successfully achieve synchronization and decode the information bits, respectively.

Table 5.3. Despreader Process Simulation Parameters

Simulation Parameter	Data Format
Synchronization Threshold	Double
Data Threshold	Double
Long Correlation	Boolean

The parameter *long correlation* has a significant impact on simulation performance. It determines the number of crosscorrelations the receiver will perform in the attempt to synchronize with arriving packets. As the receiver is designed to receive and despread $\leq N_P$ PN codes, these codes must be known to the receiver. For each code, there are k possible IC's (see Chapter II) of the shift register and k chips per code. Thus, the receiver must be configured with $N_P k^2$ codes which it can utilize to crosscorrelate with the arriving packet(s) . These codes are all generated in the INIT state and retained. Figure 5.3 offers a logical view of this persistent matrix of PN codes. IC_1 for generating each PN_i code indicates the initial condition of the shift register was set with the least significant bit as 1 (0x1), and all other bits set to 0. The remaining (up to k) codes are generated with the other initial conditions of the shift register. For example, the code for each IC_2 was generated with an initial condition of the shift register set to 2 (0x2).

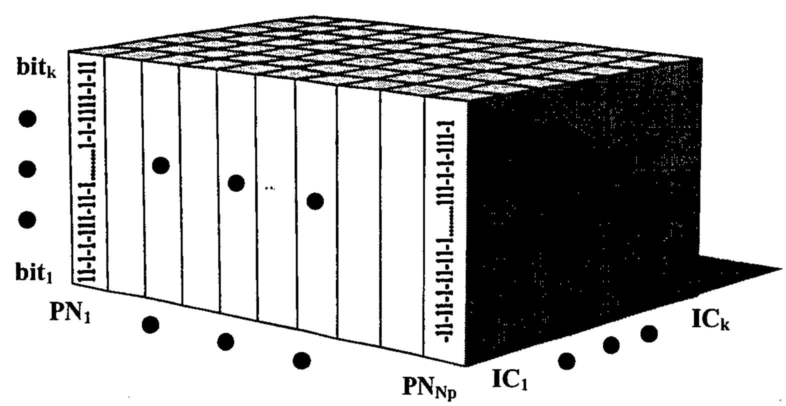

Figure 5.3. Receiver PN Code Matrix

In the SYNC state, the arriving packet is crosscorrelated with the codes of this matrix in an attempt to achieve synchronization. *Long correlation* determines if the model will execute $\leq N_P k^2$ crosscorrelations as necessary, or only PN_{i,IC_1} , i = 1.. N_P of

them. With *long correlation* disabled, a transmitter's code generated with an IC other than 0x1 should not achieve synchronization. Bit errors could prove this false. With *long correlation* enabled, the user has flexibility. The price is increased simulation time. By default, *long correlation* is disabled. The user determines the IC of the transmitter's shift register with the *User Register Load* parameter (see Chapter III).

Upon completion of the INIT state, the process is forced to the IDLE state, where it awaits incoming packets. Upon a packet's arrival, the first objective is achieving synchronization.

D. SYNC STATE

This model implements limited synchronization functionality. As OPNET is a discrete time, event driven modeling system, true modeling of spread spectrum synchronization would be prohibitive in terms of simulation time. Slotted ALOHA ensures transmitted packets arrive simultaneously. Packet synchronization is achieved as OPNET delivers the complete packet to the receiver, i.e., the receiver is not constantly sampling a received signal attempting to synchronize on a bit stream.

A realistic modeling of this receiver would require parsing the packet bit by bit and correlating first one, then two, then three and on until the time we had received the complete packet. Synchronization is a complex issue. There is phase synchronization, frequency synchronization, chip synchronization, code synchronization and others to be concerned with [14]. As OPNET delivers the complete packet to the receiver via a stream interrupt, we assume that synchronization is achieved, and the treatment of synchronization here is limited to recovering the PN code.

1. Determining Synchronization

We transition to the SYNC state on the condition PKT_RECEIVED, indicating one or more packet arrivals. From the matrix of codes generated in the INIT state, we can precisely outline the implementation of this state with pseudocode. The simplified version is as offered below. With *long correlation* disabled, **j** is fixed at 1, such that only the first sequence of each PN code will be used in the crosscorrelation process with the arriving packet:

```
Do{
  for(i ≤ N_P)
    for(j < k)
        sync = xcorr(PN_{i,j}, packet)
    end
  end
}while(sync threshold not met && more codes to check)
```

Achieving synchronization of a packet is the first step in considering the packet as successfully received. Packets that cannot be synchronized are considered lost packets and they are destroyed.

2. Multiple Access Reception

The proposed system must incorporate multiple simultaneous reception of packets. The second step in deciding successful packet reception is to determine if all simultaneously received synchronized packets have unique PN codes. Packets containing duplicate PN codes are considered collisions and destroyed.

Consider the simultaneous reception of three packets. The receiver chooses one of the packets. Assuming the receiver is able to synchronize on the first packet, we must ensure that we maintain this packet. It is a **candidate** for a successfully received packet. We cannot, however, consider it a valid received packet until all three have been synchronized and determined to have unique PN codes. If all three do contain unique PN codes, then, and only then, can we consider the three as successfully received packets.

Internally, OPNET maintains an event list comprised of events scheduled for various simulation times. The event list contains at least three events scheduled for time $t = now$; the arrival events of the three packets. We do not transition to the DECODE state until we have processed all *packet arrival* events scheduled for time $t = now$.

The model maintains each synchronized packet in a list schematically shown in Figure 5.4. A separate list is maintained for each defined PN code. If synchronization is achieved on an arriving packet, the packet is inserted in this list for persistence. Should two packets synchronize on PN_2, both packets will be inserted into the same list. For example, the list pointed to by the PN_2 list pointer will have two packets in this list, as shown in the diagram. Additionally, an OPNET interrupt labeled SYNC_ACHIEVED is scheduled to occur at time $t = now + \tau$. After all simultaneously arriving packets have

been evaluated, this process returns to the IDLE state. A SYNC_ACHIEVED interrupt will cause a transition to the DECODE state, τ seconds later.

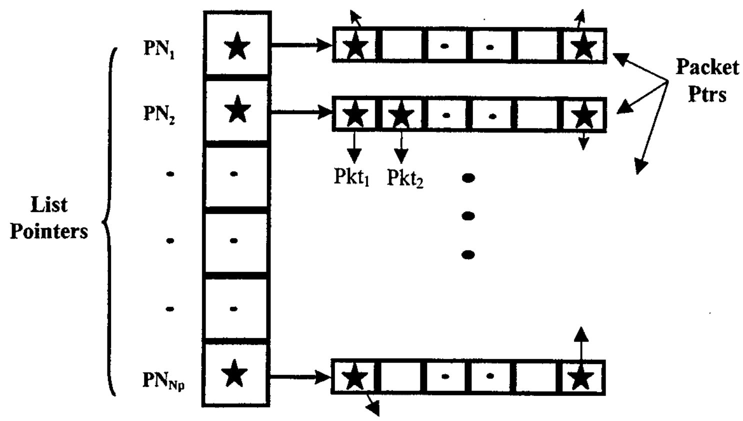

Figure 5.4. Persistent List of Synchronized Packets

3. Decoding Delay

The parameter τ referred to in the SYNC state is an induced time delay which prevents the process from transitioning directly from the SYNC state to the DECODE state. The purpose of τ is twofold. First, to provide a time gap separating the events of SYNC and DECODE. Second, to accommodate any internal variance of times maintained by OPNET for the start of reception of other packets.

The value of τ must be nominal such that packets from later slot(s) are not included in the list of synchronized packets, i.e., the current set of simultaneously arriving packets must be decoded before the next time slot. Equidistant transmitters and the MAC ensure simultaneous arrival. For transmitters at varying distances, we assume receiver feedback to ensure that all packets arrive at the beginning of a slot time.

E. DECODE STATE

This state determines packet collisions, despreads non-colliding packets, and attempts recovery of the original information. Transition to this state is conditioned on ACHIEVED_SYNC, defined as 1) synchronization achieved and 2) invocation of a SYNC_ACHIEVED interrupt.

1. Collision Decision

Packet collision is determined by evaluating the list(s) indicated in Figure 5.4. Any list(s) of PN_i maintaining $J > 1$ packets indicates duplicate codes. All J packets are destroyed.

2. Strip, Despread, Decode

For all PN_i maintaining $J = 1$ packet, the packet is considered as received successfully. Now, the receiver can move ahead to decoding and despreading the packet. Despreading and decoding are accomplished by crosscorrelating the spread information bits with the respective PN code(s) recovered in the SYNC state. A single iteration through the list in Figure 5.4 will identify all successfully received packets which can now be decoded.

Decoding results in either a recovered 1 or 0 as appropriate, or this model flags information bit errors as –1 for bits failing to reach correlation exceeding *data threshold.* Table 5.4 depicts the trivial case of $n = 3$, $k = 7$, indicating the original information bits were spread with a PN sequence of seven chips. Information bit 1 consists of the seven encoded bits shown in the first row. Information bit two follows on the second row, and information bit three follows on the third row.

Table 5.4 Decision value of Recovered Data Bits

	Spread Bits, PN Code	**Normalized Correlation**	***Data Threshold***	**Decision**
Info Bit 1 PN Code	11-1-11-11 11-1-11-11	1	0.9	0
Info Bit 2 PN Code	-1-111-11-1 11-1-11-11	-1	-0.9	1
Info Bit 3 PN Code	11-1-11-1-1 11-1-11-11	0.71	0.9	-1 (info bit error)

3. Event Canceling

The first transition to this state results in destruction of all colliding packets and despreading all successfully received packets. The final actions are traversal of OPNET's event list. All pending interrupt events labeled SYNC_ACHIEVED scheduled for time

$t = now$ are void, as the synchronized packet which scheduled this interrupt has been processed.

For example, in the case of three simultaneous successful synchronizations, there were three SYNC_ACHIEVED interrupts scheduled for time $t = now + \tau$. Time t is now equal to τ. One of the SYNC_ACHIEVED interrupts invoked this process. The other two are unnecessary. This model iterates through the event list canceling those unnecessary events.

F. SUMMARY

In this chapter, we presented the details of the modeling a spread spectrum CDMA receiver. Limited modeling of synchronization was presented. We presented in detail the implementation of multiple access mechanisms, using OPNET. The details of despreading and decoding the information bits were also presented.

VI. SIMULATION RESULTS

In this chapter we present an evaluation of the implemented system by summarizing the simulation results obtained from the model. Both the Rayleigh fading and the standard Gaussian approximation (SGA) bit error rate (BER) models are considered to study the models' channel characterization. We evaluate the multiple access capability of the receiver as a yardstick for the models' CDMA capability and functionality. This evaluation incorporates proper PN code generation, spreading, despreading, decoding and decision making.

The modeled system supports both voice and data users. As shown throughout this writing, a given user may generate more than one call of the respective type (N_D, N_S). The number of possible simulations is infinite. Complete verification of all of these aspects would be through intensive testing, simulation and evaluation. Here, we evaluate the networking aspects of this model by testing the system under two typical scenarios: 1) nodes consisting of speech users only, and 2) nodes consisting of data users only.

We demonstrate the performance of the medium access control under the implemented slotted ALOHA MAC and CDMA physical layer. The measurement of performance would be the percentage of spread spectrum utilization of the channel capacity, which we term spread spectrum utilization.

In presenting the models' evaluation, we first offer a common baseline on which the simulations were based. A summary of the simulation parameters used is provided. Next, we present the results of the BER models. The third section offers a graphical presentation of the models' CDMA capability. We conclude this chapter with an evaluation of the performance of the MAC under CDMA.

A. SIMULATION SETUP

In this section we review the simulation parameters for the model. A partial listing of the user definable parameters is given in Table 6.1. We note unspecified parameters or those that vary from the defaults as necessary.

Table 6.1. Baseline Simulation Parameters

Simulation Parameters	Data Format	Default Value
Active Sources	Integer	20 Data / 1 Voice
User Data Bits	Character String	None: User Must Specify
Slot Length	Double	0.0
Chip Rate	Double	1.2288E6 (chips/sec)
Guard Band	Integer	2 (octets)
User Register Load	Character String	None: User Must Specify
Polynomial	Character String	R(3,7)
Sync Threshold	Double	0.8
Data Threshold	Double	0.8
Long Correlation	Boolean	FALSE

As the models of [6] were developed in support of ATM research, here we chose similar packet formats as shown in Figure 6.1. The format consists of a synchronization frame of 1 octet, a 5 octet MAC header, and a 53 octet information packet. The processing gains (k) of each segment were 127, 31, and 31 respectively, giving a spread packet length of 15,400 chips. The PN codes were generated with a seven bit register, providing 18 unique PN codes and a processing gain of 127 for the synchronization frame. However, as the traffic models employed have an equivalent data rate of approximately 32 kbps, Equation (2.7) restricts k to 31. To compensate for this, we reduced the synchronization frame from four octets to one octet. Rather than spreading four octets by 31, we spread 1 octet by 127. This occupies approximately the same bandwidth, yet ensures the sequences generated for spreading the synchronization frame remain orthogonal.

Including guard band time, the resulting slot length, T_S, of the spread packet is approximately 13 ms. The subchannel capacity is given to be approximately 77 packets per second; the equivalent data rate is approximately 39.6 kbps or 1.228E6 chips per second.

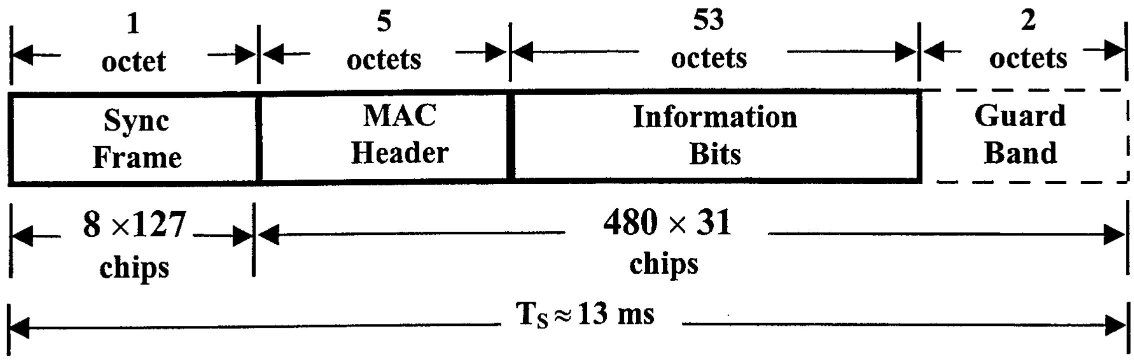

Figure 6.1. Simulation Packet Format

The channel is configured with the parameters listed in Table 6.2. As discussed in Chapter III, the *Modulation* parameter is irrelevant in this selection as we determine the BER directly without reliance on OPNETs' internally defined BER tables. However, the user is required to make a choice. The value for *ECC threshold* allows the receiver to accept essentially all packets.

Table 6.2. Transmitter/Receiver Channel Parameters

Simulation Parameters	Data Format	Default Value
Modulation	N/A	N/A – Choose Any
ECC Threshold	Double	1 (Accept all errors)
Noise Figure	Double	7 dB
Data Rate (Chip Rate)	Integer	1.2288E6
Bandwidth	Double	2 MHz
Minimum Frequency	Double	824 MHz
Spread Code	Integer	Disabled
Transmitter Power	Character String	Varied

The channel parameters closely resemble those of IS-95; see Table 9-2 of [10]. Shown are 60 subchannels per RF channel which is based on a data rate, R_b, of 9600 bps, and a chip rate, R_c, of 1.2288E6 chips per second. As the traffic sources have a data rate of approximately 32 kbps, N_P is restricted to 12. For the multiple access implementation, the receiver PN code matrix is defined using the following polynomials: R(3,7),

R(2,3,4,7), R(1,2,3,4,5,7), R(1,7), R(6,7), R(1,3,5,7), R(1,2,5,7), R(2,3,4,5,6,7), R(1,2,3,7), R(1,2,4,5,6,7), R(2,4,6,7), and R(1,2,3,5,6,7).

The required channel capacity is

$$C = \frac{N_P}{T_S} \text{ packets per second} \tag{6.1}$$

where N_P is the number of unique PN codes and T_S is the slot time in seconds. Thus, the channel capacity for the simulation is

$$C = \frac{12}{0.013} \approx 925 \quad \text{packets per second}. \tag{6.2}$$

The spread spectrum utilization of the MAC is then given by:

$$\eta = \frac{\text{Successfully Received Packets / seconds}}{C}. \tag{6.3}$$

The numerator is calculated by dividing the number of successfully received packets at the receiver by the total simulation time (seconds), where we have defined successfully received packets as those packets that were successfully synchronized and were not considered PN collisions.

B. BIT ERROR RATE EVALUATION

In evaluating the modeling of the channel, we consider the Rayleigh fading model and the SGA BER as discussed in Chapter II.

1. Rayleigh Fading Channel

The results of the Bit Error and Error Allocation pipeline stages implementation of the Rayleigh fading channel BER are shown in Figure 6.2. The curve represents the average rate of errors injected from within the Error Allocation pipeline stage. The curve shows the dependency of the BER on the variation of the received E_b / N_0. A detailed treatment of fading channels can be found in [10]. The focus here is on the modeling of

such a channel within the OPNET environment. Figure 6.2 illustrates the impact of channel fading and the linear relation between the probability of error and E_b / N_0. Figure 6.2 shows plots of both the model results and Equation (2.11); however, the results produced by the model match so closely with Equation (2.11), the two curves completely overlap each other. Throughout the simulation, the calculated BER of the Bit Error pipeline stage and the average BER from injected errors within the Error Allocation pipeline stage are quite similar to theoretical expectations.

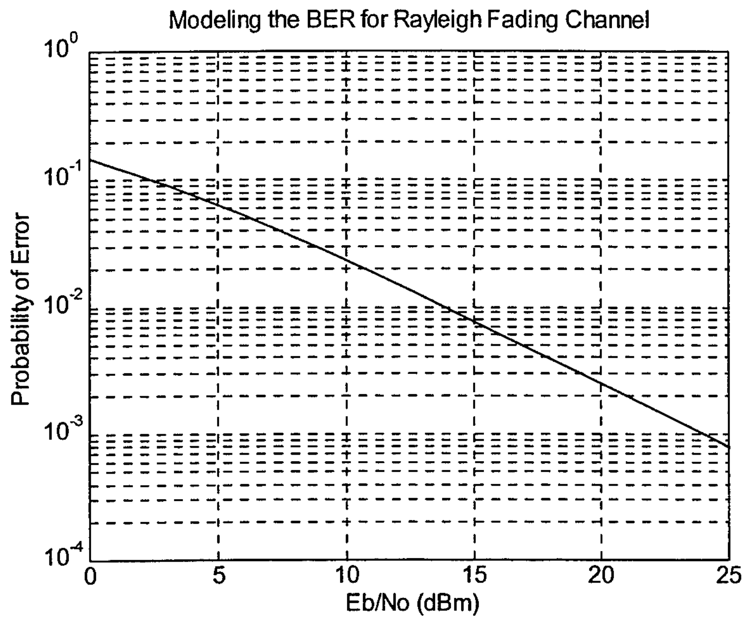

Figure 6.2. Rayleigh Fading Channel BER Simulation Results

2. Standard Gaussian Approximation

The implementation results of the SGA model are shown in Figure 6.3. Plotted are the curves for K = 3, 6, 9, and 12 nodes within the network. The solid curve plots the theoretical BER as determined in the BER pipeline stage. The dashed line curve plots the rate of errors injected into the packets in the Error Allocation pipeline stage. As discussed in Chapter II, this BER modeling is for the case of a non-fading, noise limited channel. The results indicate the BER of injected errors closely parallels the theoretical expectations.

The degradation suffered in a fading channel is apparent when comparing Figures 6.2 and 6.3. This comparison highlights the importance of model selection when

designing a communications system. As the proposed system is essentially a cellular system, the Rayleigh fading model may provide the best realistic expectations.

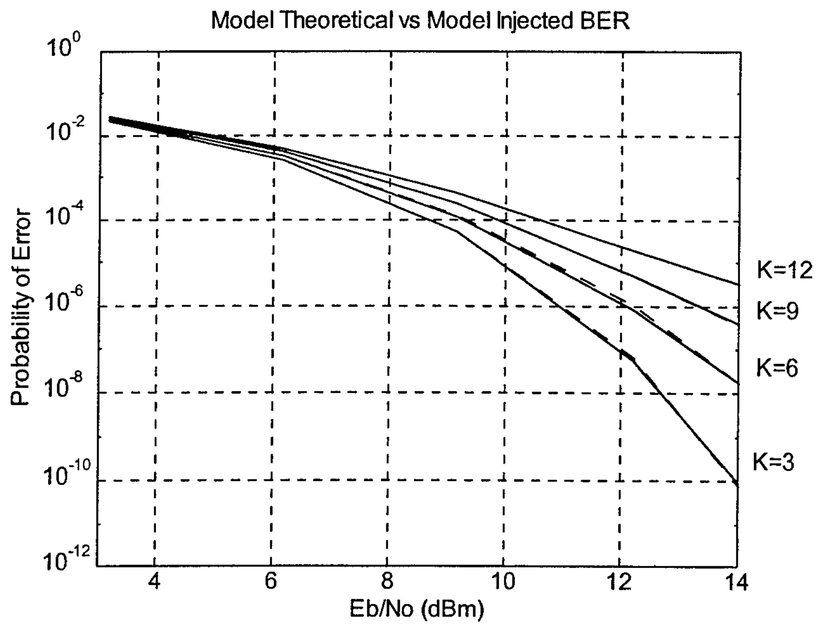

Figure 6.3. Model Implementation of Gaussian Approximation

C. CODE DIVISION MULTIPLE ACCESS SIMULATION

In this section we evaluate the CDMA capability of the model. The simultaneous reception of multiple packets is illustrated in Figure 6.4. The constant rate traffic source was employed at a rate of 10 packets per second. The upper graph shows the accumulations of the total number of packets transmitted, the total received packets, and the total that were determined as successful receptions. The lower graph magnifies the beginning of the upper graph for clarity. Shown is the case of $K = 18$. E_b/N_0 was set to 12 (dBm) such that the probability of synchronization failure was essentially zero, considering Tables 6.1 and 6.2. As shown, 12 of the 18 packets were considered collisions and were rejected.

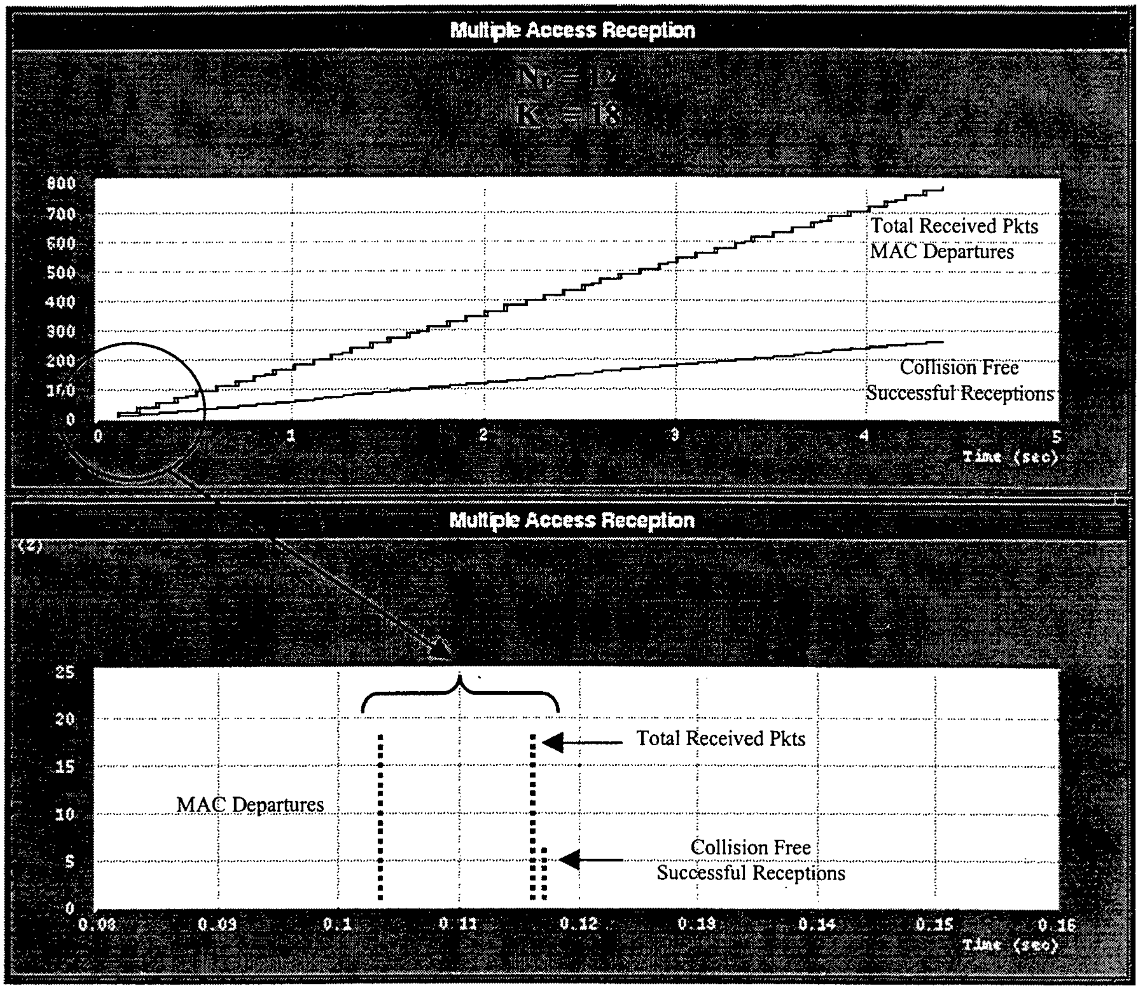

Figure 6.4. Illustration of Multiple Packet Reception

D. EVALUATION OF MAC PROTOCOL PERFORMANCE

Here we discuss the way the MAC performance (η) is measured and analyze the performance of the slotted ALOHA MAC protocol. We test two cases representing typical scenarios in the military environment: a pure speech network (representing traditional voice networks), and a pure data network (representing new networks for fire control, C^4I, battlefield dominance, etc.). In each scenario, we analyze the spread spectrum utilization of the protocol under various traffic loads.

1. Speech Traffic

The speech model generates traffic at a mean rate of 75 packets/second. Since the subchannel capacity is approximately 77 packets/second, for a single user of a given PN code, the spread spectrum utilization should be approximately 97%.

The simulation results for the pure speech scenario are shown in Figure 6.5. We see that for up to 12 users (N_P), the spread spectrum utilization, η, rises almost linearly up to approximately 97%. This is to be expected as all PN codes are unique up to this point and there are no collisions in any of the subchannels. For every additional user beyond N_P, we see dramatic degradation in η as the subchannels are no longer used by a single user. The high spread spectrum utilization of the subchannel of a single user (approximately 97%) also contributes to the degradation.

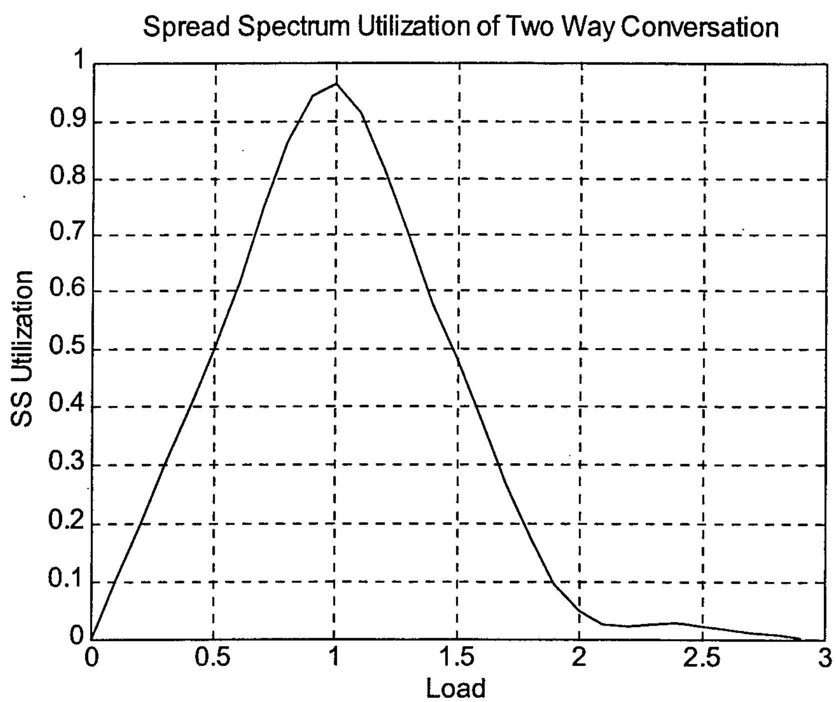

Figure 6.5. Spread Spectrum Utilization with Voice Traffic

2. Data Traffic

The data model generates traffic at a mean rate of 3.85 packets per second, per data source. The subchannel capacity remains the same as above at approximately 77

packets per second. As each node was simulated with 10 sources per node ($N_D = 10$), η of a single subchannel is expected to be $\frac{3.85 \times 10}{77}$ or approximately 50%. From Figure 6.6, consider the result for K = 12. With each node utilizing a unique PN code (thus a separate subchannel) for 50% of the time, we observe that the spread spectrum utilization is slightly less than 50%.

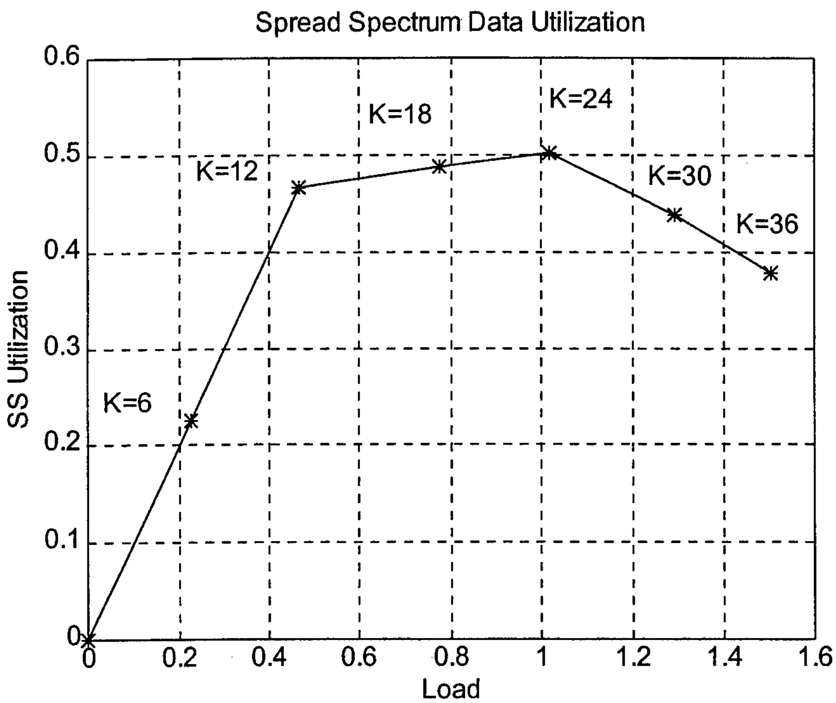

Figure 6.6. Data Utilization Under Heavy Loads

Now suppose that traffic from two different nodes is spread equally in time. As each data node multiplexes 10 bursty data streams, we can assume a Poisson behavior for the combined stream [18]. Therefore, the expected spread spectrum utilization for the case of two users per PN code is:

$$\eta^{(2)} = \Pr\{\text{only PN user 1 transmits}\} + \Pr\{\text{only PN user 2 transmits}\}$$

We have shown above that the spread spectrum utilization of each user is approximately 50%. Assuming that the traffic patterns of all users are similar, the spread spectrum utilization, η, given that two PN Codes are used, can be shown to be

$$
\begin{aligned}
\eta^{(2)} &= 2 \times \Pr\{\text{only PN user 1 transmits}\} \\
&= 2 \times \Pr\{\text{PN user 1 transmits}\} \times (1 - \Pr\{\text{PN user 2 transmits}\}) \\
&= 2 \times 0.5 \times (1 - 0.5) \\
&= 0.5.
\end{aligned}
$$

Figure 6.6 indicates that the measured results compare with the above analysis. For example, for K equals 24, the spread spectrum utilization is approximately 50%. We can easily extend this analysis to the case of 3 users per PN Code. Based on the above, the spread spectrum utilization can be shown to be

$$\eta^{(3)} = 3 \times 0.5 \times (1 - 0.5)^2 = 37.5\% .$$

From Figure 6.6, for K equals to 36 nodes, the measured spread spectrum utilization is approximately 37.5%.

E. SUMMARY

In this chapter we presented simulation results of the implementation of the proposed model in OPNET. BER performance of the RF channel has been studied for both the Rayleigh fading channel and the SGA. Spread spectrum utilization performance of the MAC protocol with an underlying spread spectrum CDMA physical layer was measured.

VII. CONCLUSIONS AND RECOMMENDATIONS

A. CONCLUSIONS

The objective of this thesis was the OPNET modeling and implementation of a CDMA system for the uplink transmission of a single cell network. To meet this, we implemented traffic generation, slotted ALOHA medium access control, spread spectrum transmission and (multiple access) reception, and the RF channel processes in OPNET. Also addressed were the issues of packet formatting, time slots within the MAC, packet spreading, bit manipulations of the packets within the BER and Error Allocation pipeline stages, and proper despreading of the packet in the receiver.

The proposed system consists of three major modules: the transmitter, the channel, and the receiver. We implemented all three modules in OPNET. As part of this effort, we were able to use some OPNET supplied modules (with minor modifications), but we developed several other modules as their functionality was not available in OPNET.

As part of the transmitter module implementation, we developed the appropriate traffic generation, slotted ALOHA MAC protocol, PN code generation, and spreading modules. The channel was implemented utilizing OPNET's fourteen stage pipeline architecture. The different pipeline stages had to be appropriately configured to realize two channel models: SGA and Rayleigh fading. The significant effort here consisted of developing the corresponding BER models and injecting errors into packets accordingly.

The receiver implementation consisted of (simplified) synchronization, packet despreading, and packet decoding modules. The functionality of these modules are not readily available in OPNET, hence they were coded and integrated into the model.

The designed system was then simulated, and some performance results were obtained. The bit error rate performance of the system was measured for both SGA and Rayleigh channel models. The measured results agree well with theoretical results. The multiple access performance of the MAC protocol under CDMA was studied by measuring the spread spectrum utilization performance. The measured results were obtained for both voice and data traffic, and they closely match with the expected theoretical performance.

B. RECOMMENDATIONS FOR IMPROVEMENTS

This modeling effort was designed to be a starting point, offering a foundation upon which the proposed communications system might be built. Future work on this model might consider the implementation and evaluation of mixed data and voice traffic, node mobility, power control issues, queuing algorithms, and receiver feedback issues as potential areas of expansion.

Based on the experience of this effort, we conclude that OPNET's modeling software is an excellent tool for wireless network design and simulation. It was a difficult task to understand and utilize OPNET's modeling tools; but once we mastered their basic usage, we found them to be indispensable for any network modeling effort.

APPENDIX A. AN OVERVIEW OF OPNET

In this appendix we provide a brief overview of the OPNET modeling software package. This appendix reviews the OPNET hierarchy, the concept of OPNET's simulation kernel, the employment of interrupts within an OPNET simulation, and the definition of an OPNET process module. The objective here is to familiarize the reader with the basic workings of OPNET. The reader requiring a more detailed or extensive treatment of OPNET's modeling software package is referred to the 12 volume set of the OPNET user manuals.

A. THE OPNET HIERARCHY

OPNET is a comprehensive engineering system capable of simulating large communications networks with detailed protocol modeling and performance analysis. OPNET is hierarchical in that it implements models in three levels: the network level, the node level, and the process level. The hierarchy can be envisioned as shown in Figure A.1.

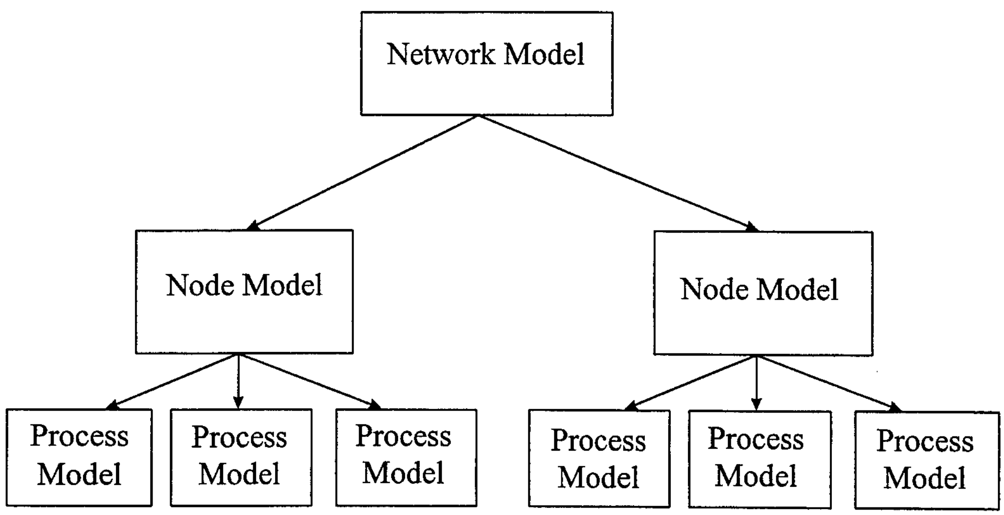

Figure A.1. OPNET Hierarchical Organization

At the network level the modeler defines the bounds of the network. The network model may represent a single network or numerous subnetworks. The network model is defined by one of the underlying node models.

The node models are defined by queues, generic processes, traffic generators, transmitters, receivers, antennas and other modules available within OPNET. Figure 5.1 offers a graphical view of the receiver node model implemented in this thesis, and is repeated here as Figure A.2. As shown, it consists of an antenna module, a radio receiver module, and two user defined processes. The PN_DESPREADER process and the SINK process are two examples of generic processes which the designer has full flexibility in designing. Implementing these generic processes is commonly where the full modeling capability of OPNET is realized.

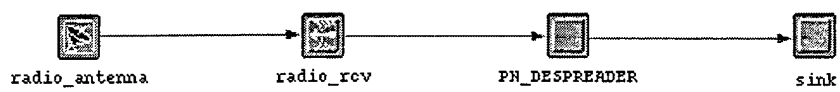

Figure A.2. OPNET Receiver Node

The arrows connecting each module of Figure A.2 play a major role in OPNET. They are referred to as *streams*. The radio_antenna module has one output stream. The radio_rcv module and the PN_DESPREADER process module have one input stream and one output stream. The SINK process module has one input stream. The interconnecting streams are the primary communication link between the different process modules within the node. To understand their use, we review OPNET interrupts and OPNET's simulation kernel.

B. INTERRUPTS AND THE SIMULATION KERNEL

OPNET's simulation kernel (SK) can be considered the master scheduler and the master clock within OPNET simulations. OPNET is a discrete time, event driven system. The time axis within an OPNET simulation is referred to as sim_time(). A sim_time() equal to 0.0 indicates the beginning of the simulation. As simulations are executed within OPNET, sim_time() advances and there are a series of events which become scheduled to occur at different times throughout the simulation. Typical events might include the sending of a packet every X number of seconds, sampling a queue size

periodically, decrementing a counter or a number of other events the designer has defined. The SK is responsible for scheduling these numerous events and ensuring they are executed at the scheduled sim_time().

The various scheduled events are typically executed by the invocation of different processes which define the process modules; the process of *despreading* is executed by the invocation of the PN_DESPREADER process module shown in Figure A.2. Processes are typically invoked into execution by one of two methods; 1) sim_time() reaches the time the process is scheduled to be invoked, and the process is automatically invoked into execution, or 2) OPNET's SK alerts the process by sending an **interrupt** to the respective process via an input stream. The interrupt delivered to the process invokes the process into execution, and the functions of the process are carried out according to the definition of the process. From the designers perspective, interrupts are the most common method of invoking processes into execution.

The focus of modeling in OPNET is in defining the process models (processes) which support and implement the node model. The process models are implemented as a Finite State Machine. The Finite State Machine is defined by completing a state transition diagram which defines the various states of the process and the transitions that interconnect each state. Figure 5.2 portrays this model's PN_DESPREAD process. Figure A.3 portrays the same process as viewed from within the OPNET graphical environment.

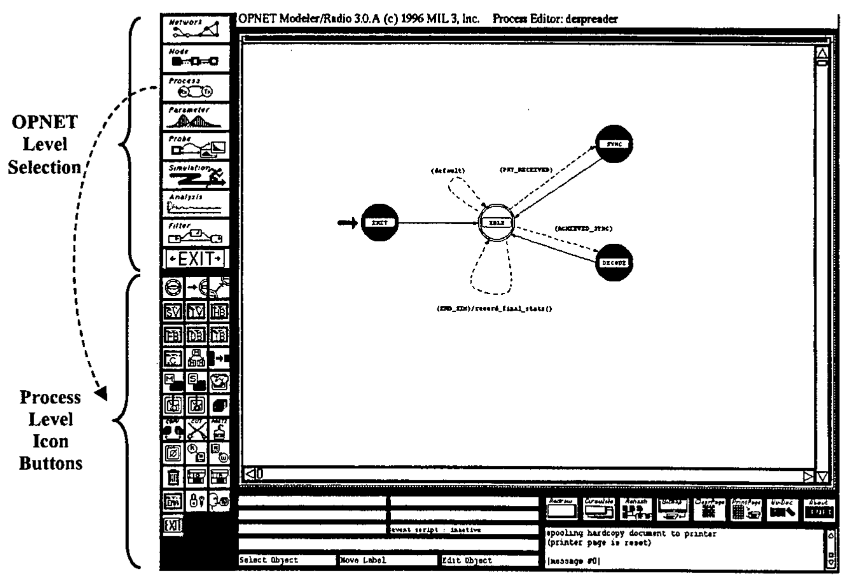

Figure A.3. OPNET View of PN_DESPREADER Process Model

C. DEFINING THE PROCESS

1. OPNET's Graphical User Interface (GUI)

As seen from Figure A.3, the PN_DESPREADER process consists of four states. The actions performed within each state are dictated by the designers source code (written in C). In completing the source code for a process, the column of icon buttons on the left side of Figure A.3 play an important role.

The icon buttons Network, Node, Process, Parameter, Probe, Simulation, Analysis, and Filter remain visible in the OPNET window throughout execution of the OPNET program. These icon buttons determine the level (Network, Node, Process) at which the designer will work, or they allow the designer to define Parameters, Probes, Simulations, perform Analysis', and define Filters. The reader interested in these areas is encouraged to consult the OPNET user manuals for a detailed description.

The icon buttons on the lower half of the column are specific to the OPNET level which is in view. For example, selecting the Process icon button from the level selection indicates the designer wishes to work at the Process level within OPNET, and is thus presented with the process level icon buttons as shown in Figure A.3. A magnified view of these icon buttons is shown in Figure A.4.

The process level icon buttons are used for writing the C source code which defines the behavior of the process. Selection of either of the HB, TV, SV, FB, DB, or TB icon buttons will invoke the OPNET editor from which the designer enters the relevant C source code. The HB icon button is used for defining the *header block* of the source code. The TV icon button similarly defines the *temporary variables* of the process. The SV icon button defines the *state variables*, and likewise FB defines the *function block*, DB defines the *debug block*, and TB defines the *termination block* for the process.

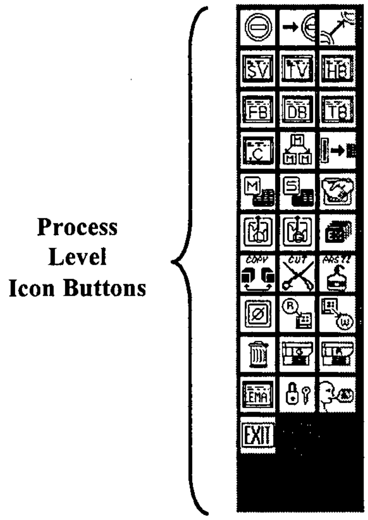

Figure A.4. Process Level Icon Buttons

The *header block* is similar to a C include.h file; **#include** <> statements and **#define** statements are likely to be coded in the header block. Temporary variables are declared in the *temporary variables* block and are not persistent variables, they exist only during the current invocation of the process. State variables are declared in the *state variables* block. They are persistent and retain their value from one invocation of a process to the next. For example, this despreader process is invoked every time a packet is received at the receiver. Any defined temporary variables are persistent only for the processing of this specific packet. Any state variables are persistent for all packets, as state variables are persistent throughout the simulation. A loop counter is an example of a common temporary variable. State variables are similar to static variables in C. Accumulators are a common example.

The *function block* is where the designer defines the various functions used in the process. The functions are typically called from one of the states, but can of course be called from another function. A common approach is to minimize the code within the states so that they are defined primarily by a series of function calls, and to off-load the bulk of the processing to the various functions defined in the function block. The *debug block* and the *termination block* will not be discussed here.

2. Forced and Unforced States

There is significance to the different color shading of the states shown in Figure A.3. States within OPNET are either *forced* or *unforced* states. Unforced states are transparent in color, forced states are opaque. A forced state is synonymous to an atomic instruction; it allows no interruptions during it's execution. A forced state controls the simulation until it has completed it's actions; it cannot be interrupted by the SK. An unforced state, on the other hand, returns control of the simulation to the SK once the process has entered an unforced state. A process which is in an unforced state is merely waiting another invocation call from the SK.

Once the SK has regained control of the simulation from the process (indicating the process is in an unforced state), the SK is free to invoke other processes of the simulation. The key difference between forced and unforced states is that there is no time lapse between entering and exiting a forced state; all actions within a forced state are essentially executed at the same time. With an unforced state, the time between entering and exiting the state is determined by the overall simulation and the different processes involved. The transparent IDLE state indicates it is an unforced state. The remaining opaque states indicate they are forced states, and will not be interrupted by the SK until they have completed the processing they are defined to perform.

As an example of the above discussion, let us consider the despreader process defined by the state transition diagram of Figure A.3. At the beginning of the simulation, the INIT state is evaluated and initialization of the process is performed. Chapter V outlines the actions performed in this particular INIT state. As the INIT state is a forced state, all actions within this state will be performed at the beginning of the simulation without any interruption from the SK, and without any elapsed time. Upon completion of initialization, the process is forced into the IDLE state. As the IDLE state is an unforced state, control of the simulation returns to the SK, and as no time lapse was incurred during the initialization, sim_time() = t_0 indicating the simulation time clock (sim_time()) has not advanced.

Upon regaining control of the simulation from the despread process, the SK checks the event list. The SK determines the next scheduled event and the time that this event is scheduled to occur. The SK will then notify the relevant process at the appropriate time. The SK's notification to a process is typically performed by delivering an interrupt to the respective process.

3. A Packet Flow Example

Consider the flow of a packet from transmission to reception. At some point in time the transmitter transmits a packet. The packet is evaluated by the pipeline stages, and the SK determines the time at which the packet is scheduled to arrive at the receiver, say $t_{arrival}$. The SK then schedules a stream interrupt (OPC_INTRPT_STREAM) to be delivered to this process at this same arrival time, $t_{arrival}$. At sim_time() = $t_{arrival}$, the SK delivers a stream interrupt to this despreader process. This delivered interrupt will invoke this process into execution (assuming the process is in an unforced state).

We have seen where the despread process performed the initialization functions at the beginning of the simulation, then transitioned to an IDLE state. Thus, at the time that the SK delivers an interrupt to this process (sim_time() = $t_{arrival}$), the despread process is 'resting' idly in the unforced IDLE state. Within the header block of this process, we have the following definitions:

```
/* Header Block Definitions */
#define SYNC_ACHIEVED  1
#define PKT_RECEIVED    (op_intrpt_type() = = OPC_INTRPT_STREAM)
#define ACHIEVE_SYNC    (op_intrpt_type() = = OPC_INTRPT_SELF &&
                         op_intrpt_code() = = SYNC_ACHIEVED)
```

The SK delivered interrupt invokes the execution of this PN_DESPREADER process. Thus, at sim_time() = $t_{arrival}$ this despread process begins execution. Upon execution, the process determines that PKT_RECEIVED evaluates to a TRUE condition (the interrupt just delivered is equal to a *stream type* interrupt) and will thus transition to the SYNC state (see Figure A.3).

As the SYNC state is a forced state, it will execute the code defined within the SYNC state without interruption from the SK. The source code implementing the SYNC state is similar to:

```
/* Sync State implementation code */
packet = get_packet_from(incoming_packet_stream);    /* get the incoming packet */
do{
  for(ix = 0; ix < upper_limit_1; ++ix) {
    for(jx = 0; jx <  upper_limit_2; ++jx) {
      correlation_value = cross_correlate(packet, pn_code);
      if(correlation_value ≥ sync_threshold) {
        achieved_sync = TRUE;
        store_packet_in_list(packet_safe_keeping_list, packet);
      } /* end if */
    } /* end inner for loop */
  } /* end outer for loop */
} while (!(achieved_sync || all_pn_codes_have_been_checked));
```

From the above block of code, we see the use of several variables. The loop counters *ix* and *jx* are temporary variables and are declared in the TV block. Likewise, *packet* is a packet pointer (Packet *) and is a temporary variable. It is also declared in the TV block. The variable *packet_safe_keeping_list* represents a linked list which stores packets that have achieved synchronization. As we need this list to be persistent, it is declared in the SV block as a state variable. The functions *get_packet_from()*, *store_packet_in_list()*, and *cross_correlate()* are defined in the FB of this process.

We see from Figure A.3 that the SYNC state has only one transition leaving the state. This transition is an unconditional transition, which is indicated by the solid line and the lack of any conditional statement above the transition. Upon completion of the actions within the SYNC state (i.e. executing the above block of code) the process unconditionally transitions back to the DECODE state. As DECODE is an unforced state, control of the simulation is then returned to the OPNET SK. The SK will again evaluate the event list, determine the next scheduled event, and send an interrupt to the respective process, and the simulation continues in a like fashion.

D. SUMMARY

Here we have provided an overview of the OPNET modeling software. We reviewed several OPNET specific terms. We noted that the OPNET simulation kernel

(SK) might be considered the master scheduler and the master clock, and we noted that the SK communicates with the different processes by sending stream interrupts to the respective process. We outlined the differentiation between forced and unforced states and reviewed how control of the simulation is passed to the process while within a forced state, and is then returned to the SK once the process returns to an unforced state. With this overview, the reader is provided a basic understanding of the OPNET modeling software package.

APPENDIX B. PIPELINE SPECIFICS

The objective of this appendix is to present an overview of the OPNET pipeline stages and review the details of the Interference, the SNR, the BER, and the Error Allocation pipeline stages. This appendix is intended as a supplement to Chapter IV and is not written to stand alone; it should be consulted as necessary after reviewing Chapter IV. We assume the reader is familiar with the OPNET modeling software and has reviewed Appendix A as necessary.

A. PIPELINE STAGES OVERVIEW

OPNET Pipeline stages are user coded (or OPNET defaults) C functions that are used to model an RF channel. Most of the stages are executed sequentially as is implied in Figure 4.1. The stages query information from packets in a variety of ways, which we will soon show. In executing several of the stages there is no time lapse incurred, i.e. the invocation of several stages are scheduled for the same time and are 'stacked' on each other in the event list. Simulation time 'stands still' during the sequential execution of these stages. The SK manages the scheduling and invocations of the stages, and the designer need only be concerned with the actual coding of the C functions to properly characterize the channel according to the user's model.

When a process sends a packet through a radio transmitter, the SK realizes the need for the pipeline stages. The SK will schedule in the event list a number of function calls to invoke each of the pipeline stages at the appropriate time. When the SK invokes a pipeline stage, the packet to be evaluated is automatically passed to the user defined C function which implements the pipeline stage.

An OPNET specific term used throughout Chapter IV is TDA, which represents Transmission Data Attribute. TDA's are fields within the packet which are used to store a variety of informational values pertaining to the packet. TDA's have a bit length of zero, meaning the length of the TDA field is not considered in any pipeline calculations. TDA's are place-holders of information. There are numerous TDA's defined within OPNET. Most of the TDA's are for the exclusive use of the SK. Most all the TDA's can be read, but only a small minority of them can be written to by the pipeline stages.

Packets do not have TDA's until they are transmitted from a radio transmitter (meaning they can be queried in the receiver process, but they don't yet exist in the

spreading process). The SK establishes and sets the appropriate values for a number of TDA's once a packet has been sent from a radio transmitter, and not until.

As pipeline stages are invoked by the SK, the pipeline stages perform their requisite calculations and insert the result(s) into the respective TDA of the packet. Later stages will often make their calculations based on the value(s) set in a particular TDA from a previous stage. Thus, the pipeline stages can be considered sequentially executed C functions which read and set values of the packet's TDA's. The inclusive set of pipeline stages allow the designer to completely model an RF channel and to report the channel's characterization via the TDA's.

B. INTERFERENCE AND SNR SPECIFICS

In this section we review the details of implementing the Interference, the SNR, the BER, and the Error Allocation pipeline stages. To properly model this proposed system, it is critical to fully understand how these pipeline stages are invoked and at what respective times they are invoked by the SK.

1. Interference Noise Calculations

The Interference Noise stage is the only pipeline stage which receives two packets from the SK. Upon invocation of this stage, the SK automatically passes both packets of interest to this pipeline stage.

Interference as defined here indicates simultaneous reception of packets. Interference is caused by the overlapping reception of packets at the receiver and occurs for a packet in two instances: 1) a packet is being received and another packet arrives, or 2) a packet arrives when another packet is being received. Figure B.1 depicts an example of three simultaneously arriving packets. In this example, this pipeline stage will be invoked three times. The reasoning follows.

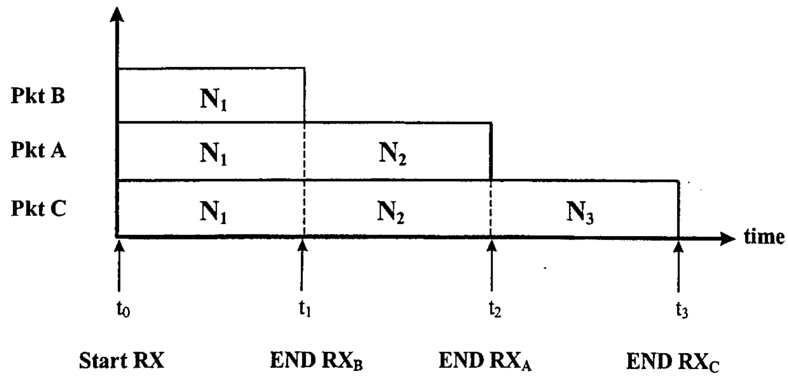

Figure B.1. Multiple Packet Reception and Respective Noise Levels

The SK orders simulation events in an arbitrary manner. There are typically several events scheduled simultaneously. In the above example, at least three events are scheduled for *time* t_0 = START_RX; the arrival events of each packet. Due to the arbitrary ordering of events, the SK must select one of the packets as the first received. Simulation runs indicate the SK selects the packet with the lowest numerical *packet id* (pkt_id) as the first arriving packet. *Packet id* is an internal attribute defined by the SK. The designer cannot set or modify this attribute.

Consider for example Figure B.1. Here we assume pkt_id(B) < pkt_id(A) < pkt_id(C). In this example, the Interference Noise stage will be invoked first with packets B and A due to packet A's interference on B, again with packets B and C due to packet C's interference on B, and finally with packets A and C due to packet C's interference on packet A.

Note also that when a packet completes reception (END_RX), the SK automatically subtracts the completely received packets accumulated noise power from any packets still being received. In the above case for example, this pipeline stage will not be re-invoked at *time* $t = t_1$. The interfering noise power TDA (NOISE_ACCUM) of packets A and C will however reflect the correct noise power due to interference: N_2 = P_r, and likewise N_3 = 0 as there is no interference during this packet segment. Equation (4.8) shows the noise power calculation employed by this model.

2. Signal-to-Noise Ratio

The SNR pipeline stage is invoked for packet a due to one of three conditions: 1) a packet **begins** reception, 2) a packet is being received and another packet arrives, or 3) a

packet is being received and another packet completes reception. Essentially upon a packets arrival at the receiver, or upon a collision.

For the example shown in Figure B.1, the SNR stage will be invoked for packet A three times (assuming the SK assigns packet B the lowest numerical packet id). Packet B will invoke the SNR stage first at *time* $t = t_0$ due to condition 1 (B arrives); a second time at *time* $t = t_0$ due to condition 2 (A arrives), and a final time at *time* $t = t_0$ again due to condition 2 (C arrives). Similarly, packet A will invoke the SNR stage first at *time* $t = t_0$ due to condition 1 (A arrives), and a second invocation at *time* $t = t_0$ due to condition 2 (C arrives). The third invocation is at *time* $t = t_1$ due to condition 3 (B completes reception). Finally, packet C will invoke the SNR stage first at *time* $t = t_0$ due to condition 1(C arrives), a second invocation at *time* $t = t_1$ due to condition 3 (B completes reception), and a third invocation at *time* $t = t_2$, again due to condition 3 (A completes reception).

Essentially, the packet is assigned a segment SNR based on the background noise during the segment interval. This models' implementation determines SNR strictly based upon the background noise N_B as shown in Chapter IV, and the differing levels of interference noise, N_I, are not utilized within this SNR pipeline stage. We include the previous discussion as it applies in the general case. Although this model employs packets of equal length, it can be employed for packets of varying lengths.

Recall upon an interfering packet completing reception, the SK will automatically subtract its received power from the noise accumulator (NOISE_ACCUM TDA) of any packets still being received. Effectively N_2 and N_3 from Figure B.1 are internally modified by the SK and the values reported in this pipeline stage will reflect the noise level due only to interfering packets.

Prior to leaving the SNR pipeline stage, a time stamp is placed on the packet. This time stamp is set in the TDA SNR_CALC_TIME and indicates the last time at which this particular packet visited this stage. In essence, it defines a packet segment of differing SNR levels. The SNR is held constant over the duration of any one segment. This parameter is necessary in the later BER and Error Allocation pipeline stages.

The primary contribution of determining the SNR is its effect in determining the bit error rate (BER) of the channel. The BER is a key benchmark in channel analysis, and we now consider this model's implementation of the BER pipeline stage.

C. BIT ERROR RATE SPECIFICS

Arguably one of the most critical components of channel characterization is the BER. The BER may be a function of several parameters, depending on the channel model used. The BER may be a function of the type of signal modulation used, a function of interference noise encountered over the channel during transmission, or a function of the amount of white noise present, or some combination. Determining the channel BER and properly injecting the errors accordingly are key focal points of this effort.

1. BER Pipeline Stage Invocation

The relevance of the discussion in the interference stage concerning the timing of pipeline stage invocation may become more obvious with what follows. Understanding the timing of the pipeline stage invocations is critical in determining the correct BER.

Similar to the interference noise stage and the SNR stage, the BER pipeline stage may be invoked for a valid packet for one of three reasons: 1) the packet **completes** reception, 2) the packet is being received and another packet arrives, or 3) the packet is being received and another packet completes reception. With the continued assumption that packet B is assigned the lowest pkt_id by the SK, and following the discussion of the interference and SNR stages, we realize the specific times each of the following packets will invoke the BER stage. Viewing Figure B.2, we can ascertain from the previous discussion and the conditions just listed that each of the three packets will

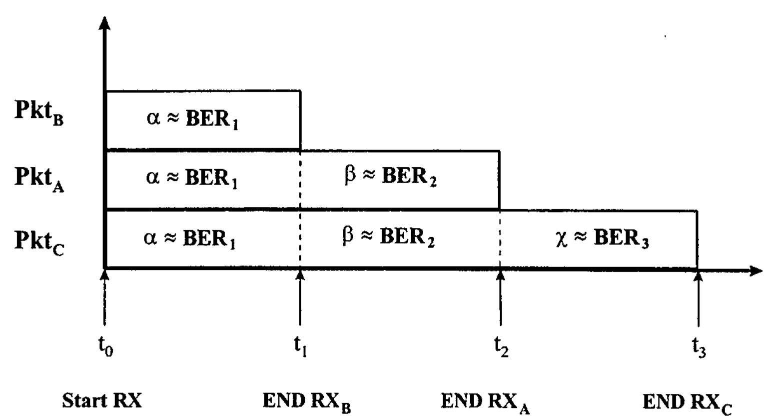

Figure B.2. BER Pipeline Stage Invocation Events

invoke the BER stage three times. Table B.1 offers a synopsis of the times and the condition (number 1, 2, or 3 above) which is met to cause the invocation. As the BER is a function of the number of K interfering users, we require the instantaneous count of interfering users on any particular packet segment.

The interference stage informs us of the number of interfering users, but only at

Table B.1. BER Pipeline Stage Invocation Times and Conditions

Packet	Invocation Times	Respective Conditions
B	t_0, t_0, t_1	2,2,1
A	t_0, t_1, t_2	2,3,1
C	t_1, t_2, t_3	3,3,1

the onset of interference. Packets may invoke the interference stage numerous times, but only at *time* $t = t_0$ (START_RX). Too, OPNET's SK will automatically subtract the power contribution (P_i) of packets completing reception from any packets still being received. As the BER is not strictly a matter of SNR, we must determine WHY this stage is invoked each time the SK invokes this pipeline stage.

2. Conditions of Invocation

We implement this stage of the pipeline by evaluating three different times for each packet; 1) *time* t = START_RX (time of starting reception), 2) *time* t = END_RX (time of ending reception), and 3) *time* t = the current simulation time. Recall that each packet maintains numerous TDA's. START_RX and END_RX are two of these, and the SK automatically determines and sets these values in each respective packet. With the intent of fully clarifying the implementation, let us walk through the BER calculation for packet B (we continue our assumption of packet B having the lowest id):

Case 1: The first time we see packet B is at *time* $t = t_0$ due to condition (2), packet A arrives at the receiver. We check the SK master clock, sim_time(), and we check the packet's START_RX. We see they are the same, thus BER = 0. This is accurate in that we are at the leading edge of the packet, the segment length = 0, and thus there are no bits in error.

Case 2: The next time we see packet B is again at *time* $t = t_0$ due to condition (2), packet C arrives at the receiver. Again, a time check indicates we are just starting reception; BER = 0.

Case 3: The final time we see packet B is at *time* $t = t_1$ due to condition (1), packet B completes reception. We check the SK master clock and we see that we are past START_RX. This indicates either we have finished reception (condition (1)), or another packet has completed reception and forced this packet to invoke this pipeline stage again (condition (3)). We must determine which of the two cases holds true.

Next we check packet B's TDA (NUM_COLLS) to determine if the packet suffered any interference from the interference stage. As the BER is a function of K (the number of multiple users), we now have K = NUM_COLLS + 1, the one being ourselves. From this, we can determine the packet's BER as shown in Chapter IV. The determined BER is set in the respective packet's TDA. Prior to leaving this stage however, we must determine why this packet invoked this stage.

If the current time is equal to END_RX, then we are here due to condition (1). No action is required, as this packet is completely received and the BER is properly determined. If the time comparison shows we are not finishing reception (current time < END_RX), then we must be here due to condition (3): another packet has completed reception. This indicates we will invoke this stage at least once more. But this packet now has one less interfering user. We must adjust the TDA (NUM_COLLS) to reflect this. We decrement NUM_COLLS by one to reflect the fact that an interfering packet has been completely received and is no longer interfering with this packet's reception. Note, the BER was determined prior to decrementing NUM_COLLS.

We can follow this logic through for packets A and C and we will find the results are the same. Rather, we offer a summary in the concise form of simple pseudocode:

```
if(time t = = START_RX)
  BER = 0
else
  BER = f(K), where K = NUM_COLLS +1     /* Rayleigh or SGA dependent */
end if
if(time t < END_RX )                     /* another packet finished   */
  decrement NUM_COLLS                    /* set it up for the next visit   */
```

Each invocation of the SNR stage defines a packet segment which will have its own segment SNR and segment BER as shown in Figure B.2. As we have seen in some cases, the segment length is zero. In others, it may be equal to the packet length. What is critical is determining the correct value of *K*.

The next objective is to inject the errors within each packet as determined from the BER stage. The Error Allocation pipeline stage is the BER injector, and is the next pipeline stage for review.

D. ERROR ALLOCATION SPECIFICS

The OPNET default error pipeline stage provides an error allocation approach which is accurate and efficient in terms of simulation time requirements. The OPNET manuals offer a detailed discussion of the algorithm used, and that approach is quite satisfactory for the general case [17]. In this model, we necessarily part from the default stage in that we must specifically test each bit of the packet against the probability of an error.

The majority of the necessary work has been accomplished in previous stages. The packet that arrives to this stage will have no less than one SNR segment and no less than one BER segment. Recall that each visit to the SNR stage defines a packet segment. Equivalently, each SNR pipeline stage invocation will cause an invocation of the BER pipeline stage which also defines a packet segment.

What remains to be done is to physically inject errors according to the determined BER. The BER was set in the packets TDA (BER) from the previous stage. This model implements this Error Allocation pipeline stage by querying the set BER and determining the effected packet's segment length. We sequence through the respective packet segment randomly inverting bits throughout the segment. The pseudocode showing this approach is presented in Chapter IV.

E. SUMMARY

In this appendix we provided an overview of the OPNET pipeline stages and reviewed the specifics of the Interference, the SNR, the BER, and the Error Allocation pipeline stages. To correctly model this system, the designer must fully understand the timing of the invocations of the different stages, as they greatly impact the model.

APPENDIX C. CONSTANT TRAFFIC GENERATOR SOURCE CODE

In this and the following appendices, we list the source code implementing this model. The code is commented throughout where appropriate.

OPNET CODE FOR CONSTANT RATE TRAFFIC GENERATOR

Header Block
Roger Standfield
November 1997
roger.standfield@mctssa.usmc.mil

```
/* Header Block Code  */

#include "mobile.h"

#define FALSE       0
#define TRUE        !FALSE

#define MAX_LENGTH_OF_USER_INPUT_BITS    10
#define ENDSIM_INTRPT       (op_intrpt_type () == OPC_INTRPT_ENDSIM)
#define     GENERATE_PKT    (op_intrpt_type () == OPC_INTRPT_SELF)
#define UPLINK_STREAM   0

/***** Function Prototypes *****/
extern void    print_the_pkt(Packet* pkptr);
int     convert_data(short array1[], char array2[]);
Packet*     create_data_packet(short data_bits_array[]);


/***** Global Variables *****/

Boolean final_scalars_written = OPC_FALSE;
int     debug;
int     runts;

int     total_source_pkts_sent;
```

State Variables Block

```
Distribution*  \ia_dist_ptr;
Objid          \my_id;
char           \user_data_bits[MAX_LENGTH_OF_USER_INPUT_BITS];
short          \data_array[DATA_BITS];
double         \gen_rate;
int            \user_data_bit_length;
int            \pkts_generated;
```

Temporary Variables Block

```
Packet      *pkptr, *data_pkptr;
double       rand_time;
int          ix;
short       *data_bit_ptr, *int_ptr;
Objid        pktid, parent id;
```

Function Block

```
Packet* create_data_packet(short data_bits_array[])
{
   short   *data_bit_ptr, *int_ptr;
   double  pkt_creation_time;
   Packet  *data_pkptr;
   Boolean  data_pkt_created = OPC_FALSE;

   ++total_source_pkts_sent;

   data_pkptr = op_pk_create_fmt("DATA_PKT");
   if(data_pkptr != (Packet *) OPC_NIL)    /* we need to ensure we have */
                                           /* a good pkt */
    {

       pkt_creation_time = op_pk_creation_time_get(data_pkptr);

       /* now get us some place to store all of this data?   */
       data_bit_ptr = (short *) op_prg_mem_alloc(DATA_BITS*sizeof(short));

       if(data_bit_ptr != (short *) OPC_NIL)    /* cool. we have a pkt */
                                                /* and a storage spot
       {
           /* now we can take the data_array[] created in the INIT   */
           /* state and do a block copy of all those data bits into  */
           /* our newly allocated storage spot. Then we'll insert.   */
           /* the address of this storage spot into our data pkt,    */
           /* effectively 'encapsulating' the data into the pkt.     */

           /* OPNET gives us the tools to make life a little easier, */
           /* just as well use them. unfortunately, this function    */
           /* call returns void, making error checking and trapping  */
           /* rather labor and processing intensive. trust the fact  */
           /* that this was tested and  verified during the          */
           /* debugging and testing stage of this development.       */
           /*  We can do a similar operation strictly with C code,   */
           /* but we're better off keeping in the OPNET environment, */
           /* as their memory allocation is customized for this      */
           /* software.                                              */

           op_prg_mem_copy(data_bits_array, data_bit_ptr,
                                  DATA_BITS*sizeof(short));

           /* now put the PO Box #, ie the address, of this memory   */
           /* location into the pkt                                  */

           if(op_pk_nfd_set(data_pkptr,"DATA", data_bit_ptr,
```

```
                    op_prg_mem_copy_create, op_prg_mem_free, DATA_BITS*sizeof(short)) ==
                    OPC_COMPCODE_SUCCESS)
           {

              if(op_pk_nfd_set(data_pkptr, "creation_time",
                        pkt_creation_time) == OPC_COMPCODE_FAILURE)
                 op_sim_message("ERROR:TRAFFIC GENERATOR:
                        send_cell_to_mac().",
                        "Failed to set pkt_creation_time in pkt. ETE
                              data may be bogus.");

              data_pkt_created = OPC_TRUE; /* we've made it all the way */
                                           /* here, set a victory flag  */
        }
        else    /* if the insertion of the address failed, free up the     */
                /* memory that was allocated                                */
        {
           op_prg_mem_free(data_bit_ptr);
           op_sim_message("ERROR:TRAFFIC GENERATOR:create_data_pkt().",
                       "FAILURE TO INSERT DATA BIT ADDRESS!");
        }
     }
     else    /* if the memory allocation for the data bits failed, the     */
             /* pkt is of no use. trash it.                                 */
     {
        op_pk_destroy(data_pkptr);
        op_sim_message("ERROR:TRAFFIC GENERATOR:create_data_pkt().",
                       "FAILURE TO ALLOCATE MEMORY FOR DATA BITS!");
     }
}


/* at this point, we either have a created data packet or we don't. If   */
/* data_pkt_create == TRUE, we're golden. if == FALSE, the something     */
/* went wrong along the way. If this is the case, allocated memory has   */
/* been released case, allocated memory has been destroyed ensuring      */
/* we don't have a build up of unused pkts.                              */
/* only need to return the results to the calling function.             */


if(data_pkt_created)
   return(data_pkptr);
else
{
   op_sim_message ("ERROR: create_data_pkt(): Failed to create Data Pkt.",
                       "");
   return( (Packet *) OPC_NIL);
}
} /* end create_data_packet() */

/*************************************************
This function records the theoretical offered load
for the entire network. Each node computes its
load value and adds it to the global load variable.
This function is called with the ENDSIM intrpt.
*************************************************/
```

```
record_theo_load ()
{
    FIN (record_theo_load ())
    /* this if condition makes sure that the theoretical  */
    /*  load value gets recorded once only and not by     */
    /*  every node                                        */
    if (final_scalars_written == OPC_FALSE)
    {
        op_stat_scalar_write ("Network Theoretical Load (bits/sec)",
                                            (double) nw_theo_load);
       final_scalars_written = OPC_TRUE;
    }
    FOUT
    } /* end function */

/*************************************************************************************/
/* Function:   convert_data(), called from INIT state.                              */
/* Inputs:     int array1[] = the array that will hold the converted data           */
/*             char array2[] = the string or character the user entered as a       */
/*             simulation parameter                                                 */
/*               parameter.                                                         */
/* Return:     int = the number of data bits determined after parsing the user      */
/*                    input                                                         */
/*                                                                                  */
/* Purpose:    This function is designed to accept a character string and convert  */
/*             this character string into bit values.  The bit values will be       */
/*             returned in the the integer array. For example, if the user entered */
/*             'A' as the data_bits  simulation parm, then '1010' will be returned  */
/*             in the integer array.                                                */
/* Assumptions:   The user simulation parms are entered upper case.                 */
/* Last Mod:   971001, r.standfield@computer.org                                    */
/*************************************************************************************/
int
convert_data(short array1[], char array2[])
{

    int     ix;

    FIN(convert_data(short array1[], char array2[]))

    for(ix=0; *array2 != '\0'; ++array2)
    {
        switch(*array2)
        {
            case '0':
                array1[ix++] = 0;
                break;
            case '1':
                array1[ix++] = 1;
                break;
            case '2':
            case '3':
                array1[ix++] = 0;
                array1[ix++] = 0;
                array1[ix++] = 1;
                if(*array2 == '2')
                    array1[ix++] = 0;
```

```
        else
            array1[ix++] = 1;
        break;
    case '4':
    case '5':
        array1[ix++] = 0;
        array1[ix++] = 1;
        array1[ix++] = 0;
        if(*array2 == '4')
            array1[ix++] = 0;
        else
            array1[ix++] = 1;
        break;
    case '6':
    case '7':
        array1[ix++] = 0;
        array1[ix++] = 1;
        array1[ix++] = 1;
        if(*array2 == '6')
            array1[ix++] = 0;
        else
            array1[ix++] = 1;
        break;
    case '8':
    case '9':
        array1[ix++] = 1;
        array1[ix++] = 0;
        array1[ix++] = 0;
        if(*array2 == '8')
            array1[ix++] = 0;
        else
            array1[ix++] = 1;
        break;
    case 'A':
    case 'B':
        array1[ix++] = 1;
        array1[ix++] = 0;
        array1[ix++] = 1;
        if(*array2 == 'A')
            array1[ix++] = 0;
        else
            array1[ix++] = 1;
        break;
    case 'C':
    case 'D':
        array1[ix++] = 1;
        array1[ix++] = 1;
        array1[ix++] = 0;
        if(*array2 == 'C')
            array1[ix++] = 0;
        else
            array1[ix++] = 1;
        break;
    case 'E':
    case 'F':
        array1[ix++] = 1;
        array1[ix++] = 1;
```

```
            array1[ix++] = 1;
            if(*array2 == 'E')
                array1[ix++] = 0;
            else
                array1[ix++] = 1;
            break;
        default:
            break;
        }
    }
    array1[ix] = -1;

    FRET(ix)
}
```

INIT State

```
/* get the objid of the generator process module */
my_id = op_id_self ();
pkts_generated        = 0;


/* read the promoted attributes at run time                                */
/*  attribute DEBUG is included for debugging/printing purposes.          */
/*  if TRUE, a bunch of printf statements throughout,                     */
/*  if FALSE, a much cleaner output                                       */

op_ima_obj_attr_get(my_id, "debug",              &debug);
op_ima_obj_attr_get(my_id, "Generation_Rate",    &gen_rate);
op_ima_obj_attr_get(my_id, "user_data_bits",     &user_data_bits);


/* convert the user data input (in hex) to array of short ints for data format */
user_data_bit_length = convert_data(data_array, user_data_bits);

/* the following loop will set the pkt full of data bits. we assume the user          */
/* entered some hex value, ie A, B, C, 1, 0,  or whatever, as the intended data bits. */
/* the following will simply repeat this input value for the length of the            */
/* data pkt. The intent is to aid in debugging and keeping track of the data.         */
/* By setting the data to a known value(s), we can give some quality assurance        */
/* as we check the program throughout and check what comes out the other end.         */
/* The user just needs to enter the bit pattern of the application data.              */
/* The coded bit pattern is repeated based on the user input bit stream at runtime    */
/* and the number DATA_BITS as defined in mobile.h                                    */

/* as well, we do this here rather than in the TRANSMIT state. It might hopefully save */
/* us a little time. In the TRANSMIT state, we simply perform a block copy of these    */
/* data bits over to a new allocated memory block, then insert the ptr pointing to     */
/* that block (address) into the pkt. We have not determined if the block copy is any  */
/* faster than the below loop, but one might suspect it certainly would be. a          */
/* nanosecond saved is a nanosecond earned maybe. The very last bit of data_array[]    */
/* was set to -1 in the convert_data() function. No big deal, as we are going to       */
/* overwrite it.now loop through the allocated array and cycle the user input bits     */
/* over the length of the array.                                                       */
```

```
for(ix = 0, int_ptr = data_array; int_ptr < data_array + DATA_BITS; ++int_ptr, ++ix)
    *int_ptr = data_array[ix % user_data_bit_length];


/* load the pkt generation distribution */
ia_dist_ptr = op_dist_load ("constant", 1.0 /  gen_rate, 0.0);
rand_time = op_dist_outcome(ia_dist_ptr);
op_intrpt_schedule_self(op_sim_time() + rand_time ,0);
```

TRANSMIT State

```
data_pkptr = create_data_packet(data_array);   /* get a created data pkt and        */
if(data_pkptr != (Packet *) OPC_NIL)           /* check it to make sure it's good   */
    op_pk_send(data_pkptr, UPLINK_STREAM);
else
    ++runts;            /* if it's hosed, adjust our global counter. we're using      */
                        /* this global counter to keep track of pkt losses other      */
                        /* than those from BER or PN collisions.                      */
                        /* ideally, runts would == 0, but it could account for        */
                        /* aborted transmissions, power hits of the transmitter,      */
                        /* gremlins in the network, whatever.                         */

/* in any case, calculate the random time for generating the next pkt    */
rand_time = op_dist_outcome(ia_dist_ptr);

/* and schedule an intrpt for the delivery of the next pkt */
op_intrpt_schedule_self (op_sim_time () + rand_time, 0);
```

APPENDIX D. SLOTTED ALOHA SOURCE CODE

OPNET CODE FOR SLOTTED ALOHA MAC

Header Block

```
#include   <math.h>
#include   "mobile.h"

#define    SUB_Q      0
#define    TX_PACKET      0
#define TO_PHYSICAL_LAYER   0
#define MIL        1E6

/* Conditional macros */
#define    PKT_ARVL        (op_intrpt_type () == OPC_INTRPT_STRM)
#define    Q_EMPTY         (op_subq_empty(SUB_Q))
#define    ENDSIM_INTRPT   (op_intrpt_type() == OPC_INTRPT_ENDSIM)
#define    BEG_SLOT       ((op_intrpt_type() == OPC_INTRPT_SELF) &&
                           (op_intrpt_code() == TX_PACKET))

/***** Global Variables       *****/

extern int       debug;    /* the three extern vars are defined in */
extern int       runts;    /* in the conversation model process    */
extern Stathandle     runts_stat_handle;

int          total_slotted_arrivals;
int          total_slotted_departures;     /* how many make it thru the Q */
double       slot_length;                  /* the time occupied by our bits */
Boolean      final_slotted_stats_written   = OPC_FALSE;
Boolean      INIT_info_written             = OPC_FALSE;
Stathandle   slotted_arrivals_handle;
Stathandle   slotted_departures_handle;


/***** Function Declarations *****/
void       print_the_pkt(Packet* pkptr);
void       record_slotted_stats();
Packet*    create_mac_pkt(short header_bits_array[]);
Packet*    add_data_to_mac_header(Packet *header_pkptr, Packet *data_packet_ptr);
```

State Variables

```
int          \packet_length;
int          \pkts_sent;
int          \guard_band;
short        \mac_header_bits_array[MAC_BITS];
double       \CHIP_RATE;
```

```
Objid        \my_id;
Boolean      \mac_pkt_sent;
Stathandle   \chan_access_local_handle;
Stathandle   \chan_access_global_handle;
```

Temporary Variables

```
short   *data_bit_ptr, *int_ptr;
double   chan_access_delay;
Packet *data_pkptr;
Packet *mac_header_pkptr;
Packet *mac_pkptr;
double   pkt_creation_time = 0.0;
```

Function Block

```
/************************************************************************
 this function is designed to print out the contents of a packet. *Packet is
 passed in. Check each field=bit of the packet, and print out the packet as
 a string of 1's and 0's.
************************************************************************/
void print_the_pkt(Packet *pkptr)
{
  int ix,data_bit, bit_length, total_size, bulk_size;

  FIN(print_pkt(pkptr))
  total_size = op_pk_total_size_get(pkptr);
  bulk_size  = op_pk_bulk_size_get(pkptr);
  bit_length = bulk_size == 0 ? total_size: total_size - bulk_size;


/* loop through each bit of the packet (excluding any bulk size field, which   */
/* is not accessible on a bit basis), extract the bit, and print it out.   */

  for(ix=0; ix < bit_length; ++ix)
  {
    if(op_pk_fd_get(pkptr, ix, &data_bit) == OPC_COMPCODE_SUCCESS)
      printf("%d", data_bit);
    else
    {
      op_sim_message("INSIDE PRINT THE PKT FNCT. COULD NOT GET THE FIELD FROM PKT", "");
      break;
    }
  }
  printf("\n");
  FOUT
}
```

```
/*******************************************************************************/
/* Function:  create_mac_pkt()                                                 */
/* Inputs:    None                                                             */
/* Return:    Packet*= A newly created pkt of format = MAC_PKT. The address of the */
/*             MAC HEADER bits is placed inside the pkt as a structure field.  */
/* Purpose:   This function is designed to add a medium access control (MAC)   */
/*        segment to a packet. This function creates a new packet, loads the first  */
/*        N bytes of this new packet with 1's, then copies over the bits of the      */
/*        received packet into this new packet. N = a user definable simulation      */
/*        parameter defaulted to 5 (bytes). The received data packet is destroyed    */
/*        within this function.                                                */
/* Last Mod:  10/97, r.standfield@computer.org                                 */
/*******************************************************************************/
Packet* create_mac_pkt(short header_bits_array[])
{
  Packet*    mac_header_pkptr;
  short      *mac_header_bits_ptr;
  Boolean    mac_pkt_created = OPC_FALSE;          /* guilty until proven innocent */

  mac_header_pkptr = op_pk_create_fmt("MAC_PKT");    /* create our new packet   */
  if(mac_header_pkptr != (Packet *) OPC_NIL)         /* if ok, continue         */
  {
    /* allocate the memory for enough mac header bits. MAC_BITS is defined in mobile.h */
    mac_header_bits_ptr = (short *) op_prg_mem_alloc(MAC_BITS*sizeof(short));
    if(mac_header_bits_ptr != (short *) OPC_NIL)
    {
      /* copy over our mac_header_bits_array into this address reserved just for us */
      op_prg_mem_copy(header_bits_array, mac_header_bits_ptr, MAC_BITS*sizeof(short));

      /* now need to put the memory address into the pkt.  */
      if(op_pk_nfd_set(mac_header_pkptr,"MAC_HEADER", mac_header_bits_ptr,
            op_prg_mem_copy_create, op_prg_mem_free, MAC_BITS*sizeof(short)) ==
                                                             OPC_COMPCODE_SUCCESS)
        mac_pkt_created = OPC_TRUE;

      else          /* else we failed to set the address in the pkt; clean up   */
      {
        op_prg_mem_free(mac_header_bits_ptr); /* pkptr & memory ok, but failed to put*/
        op_pk_destroy(mac_header_pkptr);      /* address into pkt. free up both.    */
        op_sim_message("ERROR:create_mac_pkt():\nFailed to SET mac header
                                                         address in pkt",
                    "Allocated memory is released, MAC pkt destroyed.");
      }
    }
    else
    {
      op_pk_destroy(mac_header_pkptr);  /* pkptr was ok, but no memory for header bits */
      op_sim_message("ERROR:create_mac_pkt():\nFailed to Allocate memory
                                       for header bits",
                  "MAC Pkt destroyed.");
    }
  }

  if(mac_pkt_created)
    return(mac_header_pkptr);
  else
  {
```

```
    op_sim_message("ERROR:create_mac_pkt()", "Failed to Create pkt of MAC_PACKET
                                                          format");
    return( (Packet *) OPC_NIL);      /* if we couldn't get a pkt, return a null pkptr */
  }

} /* end create_mac_pkt()  */


Packet*add_data_to_mac_header(Packet *header_pkptr, Packet *data_packet_ptr)
{
  short   *data_bits_ptr, *header_bits_ptr;
  double   pkt_creation_time;
  Boolean data_bits_added = OPC_FALSE;        /* we're always guilty aren't we? */

  if(op_pk_nfd_get(data_packet_ptr, "DATA", &data_bits_ptr)==OPC_COMPCODE_SUCCESS)
  {
    if(op_pk_nfd_set(header_pkptr,"DATA", data_bits_ptr, op_prg_mem_copy_create,
                 op_prg_mem_free, (DATA_BITS*sizeof(short))) == OPC_COMPCODE_SUCCESS)
    {
      data_bits_added = OPC_TRUE;

      /* we attempt to monitor the ETE of the original pkt. we'll pass on   */
      /* the info here. not a show stopper if it fails, just loss of some   */
      /* statistical data.                                                  */

      if(op_pk_nfd_get(data_packet_ptr, "creation_time", &pkt_creation_time))
      {
        if(!op_pk_nfd_set(header_pkptr, "creation_time",  pkt_creation_time))
          op_sim_message("ERROR:TRAFFIC GENERATOR:add_data_to_mac_header()",
                         "Failed to transfer Original PKT Creation Time.
                              ETE data may be bogus.");
      }

      op_pk_destroy(data_packet_ptr);/* the bits have been 'copied' over.   */
                                     /* no further need for this pkt        */
    }

    else             /* else we failed to set the data bits address in the  */
                     /* pkt, clean up                                       */
    {
      op_prg_mem_free(data_bits_ptr);      /* ptr is good or we wouldn't be */
                                           /* in this branch                */
      op_pk_destroy(data_packet_ptr);      /* we assume the pkptr passed in */
                                           /* was good                      */
      op_sim_message("ERROR:add_data_to_mac_header():Failed to SET
                                     data bits address in MAC pkt!",
                     "Allocated memory released, Data Pkt destroyed.");
    }
  }
  else /* else we failed to retrieve the memory address */
      /* of the data bits. destroy the data pkt        */
  {
    op_pk_destroy(data_packet_ptr);      /* again, we assume the pkptr passed   */
                                         /* in was good                         */
    op_sim_message("ERROR:add_data_to_mac_header():Failed to GET
                                 data bits address from data pkt!",
```

```
                "Data Pkt destroyed. Allocated memory for DATA_BITS is not released.");
   }

   /* if we were successful above, then we have added the pointer to the data bits   */
   /* into our MAC pkt and destroyed the data packet that brought us the data bits.  */
   /* If we had a problem we dealt with it, releasing the memory that we could get   */
   /* a pointer to, and destroyingthe data packet afterwards. Although the data pkt  */
   /* has been destroyed at this point, we  ensure we do not deallocate the memory   */
   /* where the actual data bits reside, unless of course we had a problem above,    */
   /* as mentioned. Lastly, we'll check on the results of above and if successful,   */
   /* we'll return the header_pkptr that was passed in as it now has the data bits   */
   /* added to that pkt. If we were not successful in adding the data bits, the MAC  */
   /* header pkt is of no use. We'll release the memory, if we can, of the MAC       */
   /* header bits, destroy the pkt that was passed as it serves no purpose, then     */
   /* finally return a null ptr indicating the operation add_data_to_mac_header()    */
   /* was a flop.                                                                     */

   if(data_bits_added)          /* if we successfully encapsulated the data bits,     */
                                /* we're golden                                       */
     return(header_pkptr);      /* return the passed in pkptr, with the added         */
                                /* data bits address                                  */

   else                   /* else without the data bits encapsulated, the pkt is junk.*/
   {
     if(op_pk_nfd_get(header_pkptr, "MAC_HEADER", &header_bits_ptr) ==
                                                   OPC_COMPCODE_SUCCESS)
       op_prg_mem_free(header_bits_ptr);
     else
       op_sim_message("ERROR:add_data_to_mac_header:Failed to GET
                                  MAC Header bits address from MAC pkt!",
                  "MAC Pkt destroyed. Allocated memory for MAC_BITS
                                  is not released.");

     /* regardless of deallocating the memory taken by the mac header bits, we'll    */
     /* destroy the pkt                                                               */
     op_pk_destroy(header_pkptr);
     return( (Packet *) OPC_NIL);          /* and return a null pkptr */
   }
}


void
record_slotted_stats()
{
   double normalized_load, normalized_mac_thruput;
   double channel_capacity =  NUMBER_PN_CODES / slot_length;

   /* channel capacity is determined as the number of unique PN sequences * pkts/sec,  */
   /* equating to K*pkts/sec. NUMBER_PN_CODES is defined in mobile.h. We can receive   */
   /*  one pkt per slot, thus our capacity is as shown above.                          */

   if(!final_slotted_stats_written)
   {
     op_sim_message("SLOTTED2:RECORED_SLOTTED_STATS().","Writing Final Slotted Stats");

     normalized_load    =  total_slotted_arrivals   / (op_sim_time()*channel_capacity);
```

```
        normalized_mac_thruput   =  total_slotted_departures /
                                                (op_sim_time()*channel_capacity);

        op_stat_scalar_write("Total Slotted Arrivals",      total_slotted_arrivals);
        op_stat_scalar_write("Total Slotted Departs ",      total_slotted_departures);
        op_stat_scalar_write("Normalized Load         ",    normalized_load);
        op_stat_scalar_write("Normalized MAC Thruput",      normalized_mac_thruput);
        final_slotted_stats_written = OPC_TRUE;
    }
}
```

INIT State

```
/* who am i anyways?  */
my_id = op_id_self();

/* initialize variables and set the MAC_HEADER_BITS to some values */
total_slotted_arrivals      = 0;
total_slotted_departures    = 0;
mac_pkt_sent                = OPC_FALSE;

/* set all the mac_header_bits == some value, we'll use all 1's for now    */
for(int_ptr = mac_header_bits_array; int_ptr < mac_header_bits_array + MAC_BITS;++int_ptr)
  *int_ptr = 1;

/* get the promoted attributes at run time */
op_ima_obj_attr_get(my_id,    "slot_length", &slot_length);
op_ima_obj_attr_get(my_id,    "CHIP_RATE",   &CHIP_RATE);
op_ima_obj_attr_get(my_id,    "guard_band",  &guard_band);

/* determine if the user input any value for slot_length. Default value is 0.0.   */
/* The slot length is determined as pkt_length / (chip_rate / k) where            */
/* k = pn sequence length. If the user has input different values, run the        */
/* simulation based on those values                                               */

packet_length = (SYNC_BITS*SYNC_BITS_PG) + (MAC_BITS + DATA_BITS +
                                             8*guard_band)*DATA_BITS_PG;

if(slot_length == 0.0)
   slot_length = floor(((double) packet_length / CHIP_RATE) * MIL) / MIL;
                                                  /* round it off to usecs */

if(!INIT_info_written)
{
  op_stat_scalar_write("Pkt Length", (double) packet_length);
  op_stat_scalar_write("Slot Length", slot_length);
  INIT_info_written = OPC_TRUE;
}

/* get the statistics handles */
chan_access_local_handle  = op_stat_reg("Chan Access Delay (msec)",
                      OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
chan_access_global_handle = op_stat_reg("Chan Access Delay (msec)",
                      OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
```

```
slotted_arrivals_handle    = op_stat_reg("Total Slotted Arrivals",
                          OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);
slotted_departures_handle  = op_stat_reg("Total Slotted Departures",
                          OPC_STAT_INDEX_NONE, OPC_STAT_GLOBAL);

/* Schedule a self intrpt for the first time slot        */
op_intrpt_schedule_self(op_sim_time() + slot_length, TX_PACKET);
```

QUEUING State

```
/* get the packet from the strm. */
data_pkptr = op_pk_get(op_intrpt_strm());

/* insert in q. */
op_subq_pk_insert (SUB_Q, data_pkptr, OPC_QPOS_TAIL);

/* running count of pkts offered to the MAC, essentially the Load! */
op_stat_write(slotted_arrivals_handle, ++total_slotted_arrivals);
```

TRANSMIT State

```
/* If the Q is !empty, remove the packet from the Queue, add a Header,  */
/* and collect some stats, then send the pkt to the outstream.      */

mac_pkt_sent = OPC_FALSE;

if(op_subq_empty(SUB_Q) == OPC_FALSE)
{
  data_pkptr = op_subq_pk_remove (SUB_Q, OPC_QPOS_HEAD);

  /* create_mac_pkt() will create a new pkt of format MAC_PACKET. In that function,   */
  /* we'll request a memory allocation for storage of the MAC Header bits. We add the */
  /* MAC Header there, which consists of all 1's of user definable length, as set in  */
  /* mobile.h. The pointer to this memory location is added as a structure to the     */
  /* MAC_PACKET.                                                                       */

  /* add_data_to_mac_header() will attempt to retrieve the memory ptr to the data bits */
  /* and add this pointer to the MAC_PACKET. If successful, the data pkt is destroyed. */
  /* If unsuccessful, the memory allocated for the data bits is released if possible,  */
  /* The data pkt is still destroyed, regardless of releasing the data bits memory.    */
  /* Also, if unsuccessful, the memory allocated for the mac header bits is released   */
  /* if possible, and regardless, the MAC_PKT that was created is destroyed. The end   */
  /* result is: if the data bits were successfully encapsulated within the MAC_PACKET  */
  /* we get the packet pointer we sent it, containing the added data bits. If it went  */
  /* bad, the function will clean up as much of the mess as it can, and returns a NULL */
  /* Packet pointer, flagging us that there were problems. If all went well, send the  */
  /* fully formatted and encapsulated pkt on its way. If not, chock this one up to     */
  /* gremlins in the network, and ajust our counters. We don't count this type of loss */
  /* as a pkt loss, as are pkt losses are defined as pkts failing to reach sync in the */
  /* receiver due to: 1) BER's from the channel, or 2) PN Code collisions. Thus we     */
  /* keep a gremlin count of miscellaneous pkt losses, and refer to them as 'runts'.   */
```

```
    mac_header_pkptr = create_mac_pkt(mac_header_bits_array);
    if(mac_header_pkptr != (Packet *) OPC_NIL)
    {
        mac_pkptr = add_data_to_mac_header(mac_header_pkptr,data_pkptr);
        if(mac_pkptr != (Packet *) OPC_NIL)
        {
            op_pk_send(mac_pkptr, TO_PHYSICAL_LAYER);
            mac_pkt_sent = OPC_TRUE;
            op_stat_write(slotted_departures_handle, ++total_slotted_departures);
        }
    }

    if(!mac_pkt_sent)
        ++runts;
}

/* Regardless of the Q being empty or not, schedule a self intrpt for the next    */
/* time slot                                                                       */
op_intrpt_schedule_self(op_sim_time() + slot_length, TX_PACKET);
```

APPENDIX E. PN SPREADER SOURCE CODE

OPNET CODE FOR PN DESPREADER

```
#include <math.h>
#include "mobile.h"

#define   OVER_THE_AIRWAVES   0
#define   MAC_STREAM          0
#define   MAC_ARRIVAL         (op_intrpt_type() == OPC_INTRPT_STRM) &&
                              (op_intrpt_strm() == MAC_STREAM)

/***  Global Variables       ***/

extern int    debug;
extern int    convert_data(short array1[], char array2[]);
extern void   print_an_array(char *char_string, short int_array[], int n);
extern int    runts;

/***  Function Delcarations  ***/

Packet* spread_the_pkt(short pn_sequence_array[], short sync_bits_array[], Packet
                                                  *data_pkptr);
void    generate_pn_code(short pn_code[],short polynomial_array[], short
                                                  shift_register[]);
void    shift_right(short array[]);
void    convert_poly(char array1[], short array2[]);
short   char_to_int(char char_in);
short   get_feedback(short feedback_polynomial_array[], short register_array[]);
Packet* create_spreaded_packet(short *pn_code_bit_ptr, short *sync_bit_ptr, Packet
                                                  *pkptr);
short*  allocate_me_lots_of_memory();
short*  find_mac_header_bits_in(Packet *pkptr);
short*  find_data_bits_in(Packet *pkptr);
void    spread_sync_bits(short *spreaded_bits_ptr, short *sync_bit_ptr,
                                                  short *pn_code_bit_ptr);
void    spread_mac_header_bits(short *spreaded_bits_ptr, short *mac_bits_ptr,
                                                  short *pn_code_bit_ptr);
void    spread_data_bits      (short *spreaded_bits_ptr, short *data_bit_ptr,
                                                  short *pn_code_bit_ptr);
Boolean encapsulate_spreaded_bits_with(Packet *pkptr, short *bit_ptr);
void     free_up_the_memory_from(short *mem_hog1_ptr, short *mem_hog2_ptr, Packet *pkptr);
void     transfer_data_pkt_creation_time(Packet *old_pkptr, Packet *new_pkptr);
```

State Variables

```
/* the debugging phase of this model pointed out how painfully aware we must       */
/* be of memory management. notice  most, if not all, of the structures are using  */
/* short ints. Not all are necessary, and in some cases, little  is saved. But,     */
/* to keep it all simple and a little less confusing, we've tried to standardize    */
/* on keeping most everything as short. it hopefully reduced the number of          */
/* undiscovered bugs.                                                               */

char    \user_input_register_load[REGISTER_BIT_LENGTH + 1];
char    \polynomial[3*REGISTER_BIT_LENGTH];    /* we can afford a little extra room  */
short   \shift_register_polynomial_array[REGISTER_BIT_LENGTH+1];
                                          /* REGISTER_BIT_LENGTH, SYNC_BITS,        */
                                          /* DATA_BITS, etc are defined in mobile.h */

short   \shift_register_array[REGISTER_BIT_LENGTH + 1];
short   \sync_bit_array[SYNC_BITS];
short   \pn_code_array[PN_CODE_LENGTH];
Objid   \my_id;
```

Temporary Variables

```
int        ix;
Packet    *mac_pkptr;
Packet    *spreaded_data_pkptr;
```

Function Block

```
Packet*create_spreaded_packet(short *pn_code_bit_ptr, short *sync_bit_ptr,
                                Packet *data_pkptr)
{
  int      MAC_BITS_OFFSET, DATA_BITS_OFFSET;
  short   *mac_header_bits_ptr, *spreaded_data_bits_ptr;
  short   *data_bits_ptr, *int_ptr;
  Packet  *spreaded_pkptr;
  Boolean weer_off_to_a_good_start, mission_accomplished;

  MAC_BITS_OFFSET  = SYNC_BITS*SYNC_BITS_PG;
  DATA_BITS_OFFSET = MAC_BITS_OFFSET + MAC_BITS*DATA_BITS_PG;

  spreaded_pkptr              = op_pk_create_fmt("SPREADED_PKT");
  spreaded_data_bits_ptr      = allocate_me_lots_of_memory();
  weer_off_to_a_good_start    = (Boolean) spreaded_pkptr && (Boolean)
                                                  spreaded_data_bits_ptr;
```

```
  if(weer_off_to_a_good_start)
  {
    mac_header_bits_ptr     = find_mac_header_bits_in(data_pkptr);
    if(mac_header_bits_ptr != (short *) OPC_NIL)
    {
      data_bits_ptr      = find_data_bits_in(data_pkptr);
      if(data_bits_ptr   != (short *) OPC_NIL)
      {
        int_ptr = spreaded_data_bits_ptr;
        spread_sync_bits(int_ptr, sync_bit_ptr, pn_code_bit_ptr);
        spread_mac_header_bits(int_ptr + MAC_BITS_OFFSET, mac_header_bits_ptr,
                               pn_code_bit_ptr);
        spread_data_bits(int_ptr + DATA_BITS_OFFSET, data_bits_ptr, pn_code_bit_ptr);
        transfer_data_pkt_creation_time(data_pkptr, spreaded_pkptr);
        free_up_the_memory_from(data_bits_ptr, mac_header_bits_ptr, data_pkptr);
        mission_accomplished =
                encapsulate_spreaded_bits_with(spreaded_pkptr,spreaded_data_bits_ptr);
        if(!mission_accomplished)
        {
          op_prg_mem_free(spreaded_data_bits_ptr);
          op_pk_destroy(spreaded_pkptr);
          op_sim_message("ERROR:SPREADER:create_spreaded_packet().",
                     "Data Pkt lost to gremlins.");
        }

      }
    }
  }
   else
  {
    op_sim_message ("ERROR:PN_SPREADER:create_spreaded_packet();",
          "Memory Allocation for Spreaded Bits Failed!");
  }
  return(spreaded_pkptr);
}

void
transfer_data_pkt_creation_time(Packet *old_pkptr, Packet *new_pkptr)
{
  double pkt_creation_time;
  Boolean  xfer                = OPC_FALSE;
  Boolean  both_pkts_are_legit= ((Boolean) (old_pkptr) && (Boolean) (new_pkptr) );

/* the original pkt creation time has hopefully been passed along, as this   */
/* pkt has undergone several face lifts. We'll attempt to keep passing       */
/* along the info.*/


if(both_pkts_are_legit)
  if(op_pk_nfd_get(old_pkptr, "creation_time", &pkt_creation_time))
      if(op_pk_nfd_set(new_pkptr, "creation_time",  pkt_creation_time))
      xfer = OPC_TRUE;

if(!xfer)
  op_sim_message ("SPREADER:transfer_data_creation_time(). Null pkts passed in??, go
                     figure.",
               "Failed to xfer original pkt creation time. ETE data will be bogus");
```

```
    return;
}

void
free_up_the_memory_from(short *mem_hog1_ptr, short *mem_hog2_ptr, Packet *pkptr)
{
        op_prg_mem_free(mem_hog1_ptr);
        op_prg_mem_free(mem_hog2_ptr);
        op_pk_destroy(pkptr);
        return;
}


Boolean encapsulate_spreaded_bits_with(Packet *pkptr, short *bit_ptr)
{
    Boolean spreaded_bits_address_is_set = OPC_FALSE;     /* need to prove this wrong */

    /* printf("PN_SPREADER:encapsulate_spreaded_bits().\n");
    */
    if(op_pk_nfd_set(pkptr,"SPREADED_DATA",bit_ptr,op_prg_mem_copy_create,
                op_prg_mem_free, SPREADED_LENGTH*sizeof(short)) == OPC_COMPCODE_SUCCESS)
        spreaded_bits_address_is_set = OPC_TRUE;
    else
        op_sim_message ("ERROR:PN_SPREADER:encapsulate_spreaded_bits_with().",
                "Failed to insert spreaded_bit_ptr address into pkt!");

    return(spreaded_bits_address_is_set);
}



void spread_data_bits(short *spreaded_bits_ptr, short *data_bit_ptr, short
*pn_code_bit_ptr)
{
    short   *int_ptr1, *int_ptr2;
    short   *outer_loop_upper_bound = data_bit_ptr + DATA_BITS;
    short   *inner_loop_upper_bound = pn_code_bit_ptr + DATA_BITS_PG;

    for(int_ptr1 = data_bit_ptr; int_ptr1 < outer_loop_upper_bound; ++int_ptr1)
    {
        if(*int_ptr1 == 0)
        {
            for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
                *spreaded_bits_ptr++ = *int_ptr2 == 0? -1: 1;
        }
        else
        {
            for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
                *spreaded_bits_ptr++ = *int_ptr2 == 0? 1: -1;
        }
    }
}
```

```
void spread_mac_header_bits(short *spreaded_bits_ptr, short *mac_bits_ptr,  short
*pn_code_bit_ptr)
{
   short   *int_ptr1, *int_ptr2;
   short   *outer_loop_upper_bound = mac_bits_ptr + MAC_BITS;
   short   *inner_loop_upper_bound = pn_code_bit_ptr + DATA_BITS_PG;

   for(int_ptr1 = mac_bits_ptr; int_ptr1 < outer_loop_upper_bound; ++int_ptr1)
   {
      if(*int_ptr1 == 0)
      {
         for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
            *spreaded_bits_ptr++ = *int_ptr2 == 0? -1: 1;
      }
      else
      {
         for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
            *spreaded_bits_ptr++ = *int_ptr2 == 0? 1: -1;
      }
   }
}


void spread_sync_bits(short *spreaded_bits_ptr, short *sync_bit_ptr,  short
*pn_code_bit_ptr)
{
   short   *int_ptr1, *int_ptr2;
   short   *outer_loop_upper_bound = sync_bit_ptr    + SYNC_BITS;
   short   *inner_loop_upper_bound = pn_code_bit_ptr + SYNC_BITS_PG;

   for(int_ptr1 = sync_bit_ptr; int_ptr1 < outer_loop_upper_bound; ++int_ptr1)
   {
      if(*int_ptr1 == 0)
      {
         for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
            *spreaded_bits_ptr++ = *int_ptr2 == 0? -1: 1;
      }
      else
      {
         for(int_ptr2 = pn_code_bit_ptr; int_ptr2 < inner_loop_upper_bound; ++int_ptr2)
            *spreaded_bits_ptr++ = *int_ptr2 == 0? 1: -1;
      }
   }
}
```

```
short* find_data_bits_in(Packet *pkptr)
{
  short    *mac_header_bits_ptr, *data_bits_ptr = (short *) OPC_NIL;

  if(op_pk_nfd_get(pkptr, "DATA", &data_bits_ptr) == OPC_COMPCODE_FAILURE)
  {
    if(op_pk_nfd_get(pkptr, "MAC_HEADER", &mac_header_bits_ptr) == OPC_COMPCODE_FAILURE)
    {
      op_pk_destroy(pkptr);
      op_sim_message ("PN SPREADER:find_data_bits_in():Failed to retrieve
                                   MAC & DATA bits ptr",
                      "UNABLE to Deallocate MAC_BITS + DATA_BITS bytes
                                 (short) of memory!!");
    }
    else
    {
      op_prg_mem_free(mac_header_bits_ptr);
      op_pk_destroy(pkptr);
      op_sim_message ("PN SPREADER:find_data_bits_in():Failed to retrieve
                                   DATA bits ptr",
                       "UNABLE to Deallocate DATA_BITS bytes (short) of memory!!");
    }
  }
  return(data_bits_ptr);  /* it was either set to some legit value, or is still OPC_NIL
*/
}


short* find_mac_header_bits_in(Packet *pkptr)
{
  short   *data_bits_ptr, *mac_header_bits_ptr = (short *) OPC_NIL;

  if(op_pk_nfd_get(pkptr, "MAC_HEADER", &mac_header_bits_ptr) == OPC_COMPCODE_FAILURE)
  {
    if(op_pk_nfd_get(pkptr, "DATA", &data_bits_ptr) == OPC_COMPCODE_FAILURE)
    {
      op_pk_destroy(pkptr);
      op_sim_message ("PN SPREADER:find_mac_header_bits_in():Failed to
                               retrieve MAC & DATA bits ptr",
                       "UNABLE to Deallocate MAC_BITS + DATA_BITS bytes
                               (short) of memory!!");
    }
    else
    {
      op_prg_mem_free(data_bits_ptr);
      op_pk_destroy(pkptr);
      op_sim_message ("PN SPREADER:find_mac_header_bits_in():Failed to
                               retrieve MAC bits ptr",
                       "UNABLE to Deallocate MAC_BITS bytes (short) of memory!!");
    }
  }
  return(mac_header_bits_ptr); /* it was either set to some legit value, or is still
OPC_NIL */
}
```

```
short* allocate_me_lots_of_memory()
{
  short *big_block_memory_ptr;
  big_block_memory_ptr = (short *) op_prg_mem_alloc(SPREADED_LENGTH*sizeof(short));

  return(big_block_memory_ptr);   /* this ptr is either legit, or == OPC_NIL */
}




void
generate_pn_code(short pn_code[],short polynomial_array[], short shift_register[])
{
  int  ix, jx, n, max_length;
  short  feedback;

  FIN(generate_pn_code(short pn_code[],short polynomial_array[], short shift_register[]))

  n          = REGISTER_BIT_LENGTH;
  max_length  = PN_CODE_LENGTH;

  for(ix = 0; ix < max_length; ++ix)
  {
    pn_code[ix] = shift_register[n-1];
    feedback = get_feedback(polynomial_array, shift_register);
    shift_right(shift_register);
    shift_register[0] = feedback;
  }

  FOUT
}


void
shift_right(short array[])
{
  int ix;

  FIN(shift_right(short array[]))

  for(ix = REGISTER_BIT_LENGTH-1; ix > 0; --ix)
    array[ix] = array[ix-1];
  array[0] = 0;

  FOUT
}
```

```
void convert_poly(char array1[], short array2[])
{
  int ix = 0;
  if(*array1 != 'R' && *(array1+1) != '(' )
    op_sim_end ("Error on Spreading Polynomial Value!",
                "Cannot Determine Polynomial","Exiting Simulation","");
  else
  {
    array1 += 2;                      /* skip past the 'R(' chars of the input */
    for(; *array1 != '\0'; ++array1)  /* until we hit the ending null char, '\0'      */
    {
      if(*array1 != ',' && *array1 != ')' && *array1 != ' ') /* skip commas, spaces, */
                                                             /* and the closing ')'  */
        array2[ix++] = char_to_int(*array1-1);               /* our arrays our       */
                                                             /* indexed at zero!     */
    }
     for(; ix <= REGISTER_BIT_LENGTH; ++ix)
      array2[ix] = -1;                       /* fill to the end of our array with -1's  */

  }
  return;
}

short char_to_int(char char_in)
{

  short temp;
  switch(char_in)
  {
    case '0':
      temp = 0;
      break;
    case '1':
      temp = 1;
      break;
    case '2':
      temp = 2;
      break;
    case '3':
      temp = 3;
      break;
    case '4':
      temp = 4;
      break;
    case '5':
      temp = 5;
      break;
    case '6':
      temp = 6;
      break;
    case '7':
      temp = 7;
      break;
    case '8':
      temp = 8;
      break;
    case '9':
```

```
         temp = 9;
         break;
      default:
         break;
   }
   return (temp);
}

short get_feedback(short feedback_polynomial_array[], short register_array[])
{
    int   ix;
   short  ones_count = 0;

   for(ix = 0; feedback_polynomial_array[ix] >= 0; ++ix)
      ones_count += register_array[feedback_polynomial_array[ix]];

   return(ones_count % 2);
}
```

INIT State

```
/* get this process module's own object id */
my_id = op_id_self ();

/* get the simulation input parameters */

op_ima_obj_attr_get(my_id, "user_register_load",  &user_input_register_load);
op_ima_obj_attr_get(my_id, "polynomial", &polynomial);


convert_data(shift_register_array, user_input_register_load);
convert_poly(polynomial, shift_register_polynomial_array);

/* SYNC_BITS is defined in mobile.h. It represents the number of sync bits that will   */
/* preface the application pkt. The function, spread_the_pkt() will combine the sync   */
/* bits with the received mac_pkt from the application layer, then spread this         */
/* according to the generated pn code. The sync bits are not the primary focus of      */
/* this work, thus we merely define in mobile.h the number of sync bits, and define    */
/* them to be all 1's.                                                                  */

for(ix=0; ix < SYNC_BITS; ++ix)
   sync_bit_array[ix] = 1;

/* to implement the pn sequence business, we'll create an array which will hold our    */
/* generated pn sequence. Send this array, the polynomial, and the register array to   */
/* the function. upon return, the pn_code_array will contain the generated pn          */
/* sequence.                                                                            */

generate_pn_code(pn_code_array, shift_register_polynomial_array, shift_register_array);


/* SPREADING State */
```

SPREADING State

```
/* retrieve the incoming data pkt, which we'll refer to as the mac pkt   */

mac_pkptr = op_pk_get(op_intrpt_strm ());

/* now pass in our generated pn_code_array, our sync_bit_array, and our mac_pkt   */
/* to spread_the_data(). spread_the_data() will extract our mac header bits, our  */
/* data bits, and will combine these with our sync bits into an array. Next it    */
/* will spread all of these bits according to the pn sequence in pn_code_array.   */
/* This spreaded bit stream is maintained in memory, and the pointer to it is     */
/* placed in our spreaded_pkt. The ptr to this spreaded pkt is returned from      */
/* spread_the_pkt().                                                              */

spreaded_data_pkptr = create_spreaded_packet(pn_code_array,sync_bit_array,mac_pkptr);

/* We have taken our data bits and our mac header bits and spread them into       */
/* our spreaded spreaded pkt. The memory space allocated for these bit streams    */
/* (allocated in our traffic generator and our slotted process) was deallocated   */
/*  in spread_the_pkt(). mac_pkptr, the pktptr that brought those (pointers to)   */
/* data and mac header bits  was also destroyed in in spread_the_data().          */

/* before sending, ensure we set the pkt size to the proper value. as this is a   */
/* formatted pkt of format "SPREADED_PKT", and the value of the one and only      */
/* field 'SPREADED_DATA'  is set as a structure, we effectively don't have any    */
/* pkt size. Thus, we'll have to set it here as the pipeline stages will query    */
/* this value.                                                                    */

if(spreaded_data_pkptr)
{
  op_pk_total_size_set(spreaded_data_pkptr,SPREADED_LENGTH);

  /* send the pkt out the outstream   */
  op_pk_send(spreaded_data_pkptr, OVER_THE_AIRWAVES);
}
else
  ++runts;
```

APPENDIX F. PN DESPREADER SOURCE CODE

OPNET CODE FOR PN DESPREADER

Header Block

[illegible]

```
#include <math.h>
#include "mobile.h"

#define SYNC_ACHIEVED         1
#define TO_HIGHER_LAYER       0
#define NEW_ARRIVAL           0
#define ADD_ONE               1
#define GOOD_PACKET           1
#define PN_COLLISION          2
#define VOICE_DROP            3
#define RUNT_PACKET           4
#define SYNC_FAILURES         5
#define ETE_DELAY             6
#define INFO_BIT_ERROR        7
#define MAXIMUM_VOICE_DELAY   0.040
#define TAU                   0.001

/* definitions for our state transitions   */
#define PKT_RECEIVED    (op_intrpt_type() == OPC_INTRPT_STRM)
#define ACHIEVED_SYNC   (op_intrpt_type() == OPC_INTRPT_SELF &&
                         op_intrpt_code() == SYNC_ACHIEVED)
#define END_SIM         (op_intrpt_type() == OPC_INTRPT_ENDSIM)

/* struct PN_SEQUENCE is defined to hold a particular PN sequence,        */
/* which we will generate many of in our INIT State.                      */
/* matrix_of_pn_sequences[NUMBER_PN_CODES][PN_CODE_LENGTH] is defined to  */
/* store a pn sequence for every  'm' sequence of each defined PN         */
/* characteristic polynomial.                                             */

typedef struct
   {
     short   bit_array[PN_CODE_LENGTH];
   } PN_SEQUENCE;

/***  Global Variables       ***/

extern int        debug;              /* simulation parm used for levels of debugging */
extern void    print_the_pkt();       /* defined in 'slotted' process.                */

extern int        total_slotted_arrivals;    /* these two are globals defined in the */
                                             /* slotted  process                     */
extern int        total_slotted_departures;
extern int        runts;
```

```
/* the following globals aren't needed to be globals. Defining them as such         */
/* keeps us from having topass numerous parameters just for tracking.                */
/* Alternatively, we can keep them as state vars, but we'd have to pass each every  */
/* time we needed to update. We update on every received pkt, and one time at ENDSIM*/
/* We'll vary from good programming practices and keep them global.                  */

int      total_incoming_pkts; /* total pkts of any kind reaching our rx              */
int      total_good_pkts;     /* total pkts that passed sync and weren't PN coll's */
int      total_voice_drops;   /* total lost = sum (failed sync + PN collisions      */
int      total_pn_collisions; /* updated in DECODE state                             */
int      total_sync_failures;
int      total_info_bit_errors;

double   accumulated_ete_delay;
double   accumulated_THEO_BER;      /* these are all TDA values that are set in the    */
double   accumulated_ACTUAL_BER;    /* pipeline stage. EbNo, Rayleigh, & EbNo_BER are */
double   accumulated_NUM_ERRORS;    /* added TDA's (defined in mobile.h). these are    */
double   accumulated_EbNo;          /* some of the values we are wanting to track      */
double   accumulated_RAYLEIGH_SNR;  /* and monitor.  See the *.ps.c files for more info*/
double   accumulated_EbNo_BER;

static double minimum_ete_delay = 100.0;
static double maximum_ete_delay = 0.0;

Stathandle  total_incoming_pkts_stathandle;    /* they all need to be registered, kinda */
                                               /* like checking in at the probe desk.   */
Stathandle  total_good_pkts_stathandle;
Stathandle  ete_delay_stathandle;
Stathandle  current_pn_collisions_stathandle;

/* the following is a matrix of polynomials that will be used to generate           */
/* a PN sequence.                                                                    */

int     code_matrix[NUMBER_PN_CODES][REGISTER_BIT_LENGTH] =
                 { {3, 7, -1, -1, -1, -1, -1},
                   {2, 3,  4,  7, -1, -1, -1},
                   {1, 2,  3,  4,  5,  7, -1},
                   {1, 7, -1, -1, -1, -1, -1},
                   {6, 7, -1, -1, -1, -1, -1},
                   {1, 3,  5,  7, -1, -1, -1},
                   {1, 2,  5,  7, -1, -1, -1},
                   {2, 3,  4,  5,  6,  7, -1},
                   {1, 2,  3,  7, -1, -1, -1},
                   {1, 2,  4,  5,  6,  7, -1},
                   {2, 4,  6,  7, -1, -1, -1},
                   {1, 2,  3,  5,  6,  7, -1}  };


/***  Function Delcarations  ***/

/* an explanation of each function is offered in the function block where the   */
/* function is actually coded. They are grouped by the state they are most       */
/* applicable to.                                                                */

/*** INIT State Functions ***/
void   set_register_to_first_state(short array[], int number_of_elements);
void   generate_pn_sequence(short pn_code[], short array[], int jx);
```

```
short   find_feedback(short register_array[], int row);
void    right_shift(short array[]);
void    get_next_register_state(short array[], int number_of_elements);


/*** SYNC State Functions ***/
short*    process_the_new_guy(Packet *pkptr);
void      look_at_stats_of_pkt(Packet *pkptr);
double    extract_data_pkt_creation_time(Packet *pkptr);
double    correlate_sync_bits(short *bit_ptr, short *sequence_bits_ptr);
Boolean   tag_and_store_this_pkt(Packet *pkptr, short *bit_ptr, List *list_ptr);

/*** DECODE State Functions ***/
void      empty_the_list_of_junk_pkts(List *junk_list_ptr, int number_of_junk_pkts);
int       correlate_data_bits(Packet* incoming_pkptr, short* pn_sequence_bit_ptr, short
                              data[],double threshold);
Packet*   packetize_recovered_data(short *data_bit_array, int n_data_bits);


/*** END_SIM Function, Misc Funtions and functions used by all States ***/
void    update_stats(int which_stat, float by_how_many);
void    update_accumulators(Packet *pkptr);
void    clean_up_event_list();
void    clean_up_the_mess(Packet *pkptr, short *memory_ptr);
void    record_final_stats();
void    print_an_array(char *char_string, short int_array[], int n);
```

State Variables

```
int       \max_number_of_shift_register_states;
int       \max_number_of_correlations_to_compute;
short     \pn_sequence_array[PN_CODE_LENGTH];
short     \shift_register[REGISTER_BIT_LENGTH];
short     \recovered_poly_array[REGISTER_BIT_LENGTH];
short*    \spreaded_bits_ptr;
short*    \array_of_ptrs_to_pn_sequences[NUMBER_PN_CODES];
double    \sync_threshold;
double    \data_threshold;
List*     \array_of_ptrs_to_packet_list[NUMBER_PN_CODES];    /* as defined in mobile.h */
Objid     \my_id;
Boolean   \long_correlation;
PN_SEQUENCE \matrix_of_pn_sequences[NUMBER_PN_CODES][PN_CODE_LENGTH];
```

Temporary Variables

```
Objid       temp_id;

int          ix, jx;
int          num_events;
int          number_of_data_bits;
int          number_of_packets_in_list;
short        data_array[DATA_BITS];
double       sync_correlation = 0.0;
Boolean      achieved_sync        = OPC_FALSE;
Boolean      all_PN_codes_checked    = OPC_FALSE;
Boolean      safe_keeping       = OPC_FALSE;
```

```
Packet      *spreaded_data_pkptr;
Packet      *recovered_data_pkptr;
Packet      *received_pkptr;
```

Function Block

```
/**************************************************************************************/
/* Function:   set_register_to_first_state(), called from SYNC state                 */
/* Inputs:     int array[] = our shift register                                       */
/*        number_of_elements = number of elements in integer array = shift register  */
/* Return:     void                                                                   */
/*                                                                                    */
/* Purpose:    This function is designed to accept an array of integers representing */
/*             a shift register. The function will loop through the array, setting   */
/*             each element equal to zero, and set the LSB equal to one.             */
/* Last Mod:   08/97, r.standfield@computer.org                                       */
/**************************************************************************************/
void
set_register_to_first_state(short array[], int number_of_elements)
{
  int  ix, n = number_of_elements;

  for(ix = 0; ix < n-1; ++ix)        /* loop thru and set n-1 of them to zero */
     array[ix] = 0;

  array[ix] = 1;                /* set the LSB == 1     */
  return;
} /* end of: set_register_to_first_state() */

/**************************************************************************************/
/* Function:   generate_pn_sequence()                                                 */
/* Inputs:     Packet* = pointer to pkt that will hold the generated pn sequence      */
/*        array[]= used to represent the shift register for generating the sequence.*/
/*               This array is loaded with it's initial condition = 0x1 from         */
/*               procedure, set_register_to_first_state() called from the SYNC state */
/*        int i  = a row index to code_matrix[][] defined in the Header Block. This  */
/*               index will aid in determining which polynomial is being looked at.  */
/*        p_array[] = an integer array that will be set from calls within this       */
/*                    function.                                                       */
/* Return:     None                                                                   */
/*                                                                                    */
/* Purpose:    This function is designed to generate a PN sequence based on an       */
/*             initial cond ition. This function is called from the SYNC state,      */
/*             and can be called several times. Initially, array[] is set to the     */
/*             first initial condition, 0x1. The first bit of the PN sequence based  */
/*             upon the LSB of the shift register, = array[N] where N = 9 if array[] */
/*             represents a 10 bit register. The resulting sequence bit will be      */
/*             either 1 or -1, dependent on this LSB. Next, the feedback value is    */
/*             determined, based upon the characteristic polynomials defined in      */
/*             code_matrix[][].After obtaining the feedback (a different function    */
/*             call), the shift register is shifted right one bit (another function  */
/*             call), and the feedback value is placed into the MSB of the shift     */
/*             register.                                                              */
/* Last Mod:   10/97, r.standfield@computer.org                                       */
/**************************************************************************************/
```

```
void generate_pn_sequence(short pn_code[], short array[], int jx)
{
  int  ix, max_length, n = REGISTER_BIT_LENGTH;      /* REG..LENGTH defined in mobile.h
  */
  short  feedback;

  max_length = (int) pow(2,n) - 1;

  /* loop enough times to fill out the PN sequence     */
  for(ix = 0; ix < max_length; ++ix)
  {
    pn_code[ix] = array[n-1] == 0? -1: 1;
    feedback = find_feedback(array,jx);
    right_shift(array);
    array[0] = feedback;
  }
} /* end of: generate_pn_sequence()  */

/*************************************************************************************/
/* Function:  find_feedback(), called from generate_pn_sequence()                   */
/* Inputs:    register_array[] = the shift register array                           */
/*       row = the row index of code_matrix[][] (defined in the header block) that  */
/*         we are currently looking at. This value was passed to                    */
/*         generate_pn_sequence()from the SYNC state, and carried into this function.*/
/*************************************************************************************/
short find_feedback(short register_array[], int row)
{
   int   ix;
  short  ones_count;
  ones_count = 0;

  for(ix = 0; code_matrix[row][ix] >= 0; ++ix)
    ones_count += register_array[code_matrix[row][ix] - 1];

  return(ones_count % 2);

} /* end of: find_feedback() */

/**********************************************************************************/
/* Function:  right_shift(). Called from generate_pn_sequence().                  */
/* Inputs:    array[] = the integer array representing the shift register         */
/* Return:    None                                                                */
/*                                                                                */
/* Purpose:   This function is designed to accept an integer array of length =    */
/*       REGISTER_BIT_LENGTH as defined in mobile.h. The function will shift      */
/*       each value of the register one place (bit) to the right, and upon        */
/*       completion, will set the first bit of the register = the MSB, equal      */
/*       to zero. This value is overwritten in the calling function, but to       */
/*         ensure it has a value and is not an unknown, we set it here.           */
/* Last Mod:  9/97, r.standfield@computer.org                                     */
/**********************************************************************************/
```

```
void
right_shift(short array[])
{
  int ix;

  FIN(right_shift(short array[]))

  /* loop through N-1 times, and shift each bit over by one */
  for(ix = REGISTER_BIT_LENGTH-1; ix > 0; --ix)    /* start at the end, work back     */
    array[ix] = array[ix-1];
  array[0] = 0;                         /* be safe, set the first bit (MSB) */

  FOUT
} /* enf of: right_shift() */

/*************************************************************************************/
/* Function:  get_next_register_state(), called from SYNC state if called at all    */
/* Inputs:    int array[] = the array representing the shift register.              */
/*        number_of_elements = number of elements in the array                      */
/* Return:    None                                                                   */
/*                                                                                   */
/* Purpose: This function is designed to accept an array of integers representing   */
/*   a shift reg.The function works as a counter. It will determine the next state  */
/*   of the register, based upon the current state. If the current state == 0001,   */
/*   next state == 0010.The register is assumed to be used with generating maximum  */
/*   length sequences for  generating a PN sequence for Spread Spectrum             */
/*   applications. The state of the register should never reach the ALL ZEROS state,*/
/*   or the generated m sequence would result in nothing but zeros  from this point. */
/*   Thus, an all zeros check is incorporated. If  found in this state, the retister */
/*   is set to state ONE, ie state == 00001. If found in the ALL ONES state, the     */
/*   register (counter) will roll over. ie if current state == 1111, next           */
/*   state == 0001.                                                                  */
/*   The function is designed to work on any length of array, providing the correct */
/*   number of elements is properly passed in.                                      */
/* Assumptions:  array is of short minimum n = 3 elements. If not, for(j=i+2...)     */
/*      loop will be out of bounds                                                   */
/*                                                                                   */
/* Last Mod:  08/97, r.standfield@computer.org                                      */
/*************************************************************************************/
void
get_next_register_state(short array[], int number_of_elements)
{
  int     ix, jx;
  short   all_ones;
  int  zeros_check = 0, n=number_of_elements;

  for(ix = 0; ix < n-1; ++ix)
  {
     all_ones = array[ix+1];      /* initialize with an array element, any will do*/
     for(jx=ix+2; jx < n; ++jx)
    all_ones &= array[jx];        /* we only need to know if all elements after    */
                                      /* the i'th element are '1' or '0'.          */
     array[ix] ^= all_ones;           /* the i'th element = 1 on XOR of two        */
                                      /* conditions:                 */
  }                                   /* either it's already a one, or all following  */
                                      /* elements are '1' and it's time to 'roll'.  */
```

```
   array[n-1] ^= 1;               /* toggle the LSB each time.            */

   for(ix=0; ix < n; ++ix)        /* loop through array, checking for all zeros   */
      zeros_check |= array[ix];

   if(zeros_check == 0)           /* if all bits = 0, zeros_check = 0. set   */
     array[n-1] = 1;              /* next state to the 000....1 state        */

   return;
} /* end of: get_next_register_state() */


short* process_the_new_guy(Packet *pkptr)
{
   int     packet_length;
   double  ete_delay;
   Boolean processed_ok  = OPC_FALSE;
   short   *bit_ptr      = (short *) OPC_NIL;

   if(debug == VERIFY_NUMBERS || debug == SHOW_ALL)
            look_at_stats_of_pkt(pkptr);

   ete_delay = op_sim_time() - extract_data_pkt_creation_time(pkptr);
   minimum_ete_delay = minimum_ete_delay < ete_delay? minimum_ete_delay: ete_delay;
   maximum_ete_delay = maximum_ete_delay > ete_delay? maximum_ete_delay: ete_delay;

   if(ete_delay > MAXIMUM_VOICE_DELAY)
     update_stats(VOICE_DROP, ADD_ONE);

   update_accumulators(pkptr);
   update_stats(ETE_DELAY, ete_delay);
   update_stats(NEW_ARRIVAL, ADD_ONE);

   if(op_pk_nfd_is_set(pkptr, "SPREADED_DATA")){
     if(op_pk_nfd_get(pkptr, "SPREADED_DATA", &bit_ptr))
       processed_ok = OPC_TRUE;
   }

   if(!processed_ok){
     clean_up_the_mess(pkptr, bit_ptr);
     op_sim_message ("DESPREADER2:SYNC STATE:process_the_new_guy().",
                "Failed to get the memory address for the SPREADED BITS. Runts += 1");
     update_stats(RUNT_PACKET, ADD_ONE);
   }

   return(bit_ptr);    /* null or otherwise, depending on op_pk_nfd_get() */

} /* enf of: process_the_new_guy() */
```

```
void
look_at_stats_of_pkt(Packet *pkptr)
{
  Objid  radio_xmtr_id, xmtr_id;
    char  xmtr_name[10];
  char spreading_code[20];

  /* get the id of the transmitter object which sent this pkt      */
  radio_xmtr_id =  op_td_get_int(pkptr, OPC_TDA_RA_TX_OBJID);

  /* now get the parent of this radio xmtr, which = the node that  */
  /* sent this pkt.                                      */
  xmtr_id = op_topo_parent(radio_xmtr_id);

  /* if this xmtr node has a name attribute, see who sent this pkt */
  if( op_ima_obj_attr_exists(xmtr_id, "name") == OPC_TRUE)
  {
    op_ima_obj_attr_get(xmtr_id, "name", xmtr_name);
    op_ima_obj_attr_get(xmtr_id, "polynomial", spreading_code);
    printf("Pkt came from      = %s.\n", xmtr_name);
    printf("Pkt Spreading Code = %s.\n", spreading_code);
    printf("Pkt from %s, id = %d, Start TX = %f.\n", xmtr_name, op_pk_id(pkptr),
                       op_td_get_dbl(pkptr, OPC_TDA_RA_START_TX));
    printf("Pkt from %s  id = %d, End   TX = %f.\n", xmtr_name, op_pk_id(pkptr),
                       op_td_get_dbl(pkptr, OPC_TDA_RA_END_TX));
    printf("Pkt from %s  id = %d, Start RX = %f.\n", xmtr_name, op_pk_id(pkptr),
                       op_td_get_dbl(pkptr, OPC_TDA_RA_START_RX));
    printf("Pkt from %s  id = %d, End   RX = %f.\n", xmtr_name, op_pk_id(pkptr),
                       op_td_get_dbl(pkptr, OPC_TDA_RA_END_RX));

  }
  else
    op_sim_message("NO NAME ATTRIBUTE of transmitter for this pkt!\n", "");

} /* end look_at_stats_of_pkt()*/


double
extract_data_pkt_creation_time(Packet *pkptr)
{
  double pkt_creation_time   = 0.0;
  Booleani_got_the_time    = OPC_FALSE;

  /* the original pkt creation time has hopefully been passed along, as this   */
  /* pkt has undergone several face lifts. We'll attempt to extract and return */
  /* the info here.                                                            */

  if(op_pk_nfd_is_set(pkptr, "creation_time")){
    if(op_pk_nfd_get(pkptr, "creation_time", &pkt_creation_time))
      i_got_the_time = OPC_TRUE;
  }

  /* if we didn't successfully extract the pkt creation time, warn the user */
  if(!i_got_the_time)
    op_sim_message ("DESPREADER:extract_data_pkt_creation_time():",
            "Failed to extract original pkt creation time. ETE data is bogus.");
```

```
   return(pkt_creation_time);

} /* end of: extract_data_pkt_creation_time() */

/*************************************************************************************/
/* Function:   correlate_sync_bits(), called from the SYNC state.                   */
/* Inputs:     incoming_pkptr  = the pkt received in the receiver                   */
/*             pkptr2 = the pkt containing a generated PN sequence                  */
/*             num_of_sync_bits = a simulation parameter, globally declared in the  */
/*                        header block                                              */
/* Return:     the correlation value obtained from the cross correlation of         */
/*             two sequences                                                        */
/*                                                                                  */
/* Purpose:    This function is designed to cross correlate two binary sequences. The */
/*             first sequence is from the passed in packet, which is the pkt received */
/*             in the receiver. The second sequence is a generated PN sequence, based */
/*             upon a particular poly nomial. This function may be called numerous   */
/*             times in an attempt to achieve synchronization with the incoming pkt. */
/*                                                                                  */
/* Last Mod:   9/97, r.standfield@computer.org                                      */
/*************************************************************************************/
double correlate_sync_bits(short *bit_ptr, short *sequence_bits_ptr)
{
   int     ix;           /* SYNC_BITS and PN_CODE_LENGTH are defined in mobile.h */
   short  *int_ptr1, *int_ptr2;
   static  int place_holders[3][3] = {  { 1,  0, -1},
                                        { 0,  0,  0},
                                        {-1,  0,  1} };

   int     sum = 0;

   int_ptr1 = bit_ptr;
   for(ix=0; ix < SYNC_BITS; ++ix)
   {
     for(int_ptr2=sequence_bits_ptr; int_ptr2 < sequence_bits_ptr + SYNC_BITS_PG;
                                                     ++int_ptr1, ++int_ptr2)
        sum += place_holders[1 + *int_ptr1][1 + *int_ptr2];

   }

   return((double) sum / (SYNC_BITS_PG * SYNC_BITS)); /* our normalized corr.value */

} /* end of: correlate_sync_bits() */


Boolean
tag_and_store_this_pkt(Packet *pkptr, short *bit_ptr, List *list_ptr)
{
    Boolean its_in_the_bank = OPC_FALSE; /* its in the mail, and I luv you no .... */

                        /* op_pk_nfd_set() will abort the program if passed a null */
                        /* pkptr. it's worth a couple of machine instructions to check. */
   if(pkptr != (Packet *) OPC_NIL){
     if(op_pk_nfd_set(pkptr,"SPREADED_DATA", bit_ptr, op_prg_mem_copy_create,
           op_prg_mem_free, (SPREADED_LENGTH*sizeof(short))) == OPC_COMPCODE_SUCCESS)
     {
       /* if we were able to reinsert the ptr to our spreaded bits in the pkt,      */
```

```
      /* then we can insert this pkt in a list we are maintaining. Unfortunately,  */
      /* op_prg_list_insert()returns void. It will abort the run if passed an       */
      /* illegal list pointer or malformed list (whatever that is, I can think of  */
      /* several I guess). Lacking any complete and exhaustive method to ensure     */
      /* ensure this doesn't happen, we at least(hopefully) reduce the probability */
      /* by checking to ensure the pointer is !null before making the call.        */

      if( list_ptr != (List *) OPC_NIL){
        op_prg_list_insert(list_ptr, pkptr, OPC_LISTPOS_TAIL);

        /* since op_prg_list_insert() returns void, we COULD reaccess it to        */
        /* ensure it went smoothly, i.e. if(copy_of_pkptr == op_prg_list_access    */
        /* (TAIL))..., but we could be checking for errors 'till the cows come     */
        /* home. flag it TRUE and let's get out of here.                           */

        its_in_the_bank = OPC_TRUE;
      }
      else{
        /* else for whatever reason, the list * is null, leaving no place for the pkt */
        clean_up_the_mess(pkptr, bit_ptr);
        op_sim_message ("DESPREADER:tag_and_store_this_pkt().",
                "List* is Null. No safekeeping for pkt, it's gone!");
      }
    }
    else{
      /* bit_ptr points to our spreaded data bits. if we failed to insert this     */
      /* address back  into our pkt, the pkt is of limited if any use. we do still */
      /* have the ptr to memory where all of these bits are stored. we could       */
      /* access them, but we have no 'train' to put them on. we'll just evict them */
      /* by freeing up the memory they are holding, then we'll take the broke down */
      /* Chevy they were supposed to ride and melt it down for scrap metal. No     */
      /* need to fool with the list, as we have not manipulated it in anyway.      */

      clean_up_the_mess(pkptr, bit_ptr);     /* clean up a little and alert the user */
      op_sim_message("ERROR:DESPREADER2:from SYNC state:tag_and_store_this_pkt().",
                "Failed to reinsert the spreaded bits ptr address into the pkt.
                    Data is Lost.");

      /* but don't screw with the list or the list ptr. it's fine. the list        */
      /* just doesn't grow.                                                        */
    }
  }
  else{
    /* else if the pkptr passed in was null, don't waste our time. clean house,    */
    /* same as above                                                               */

    if(bit_ptr != (short *) OPC_NIL)  /* if they passed us a null pkptr, most      */
                                      /* likely this is null                       */
      op_prg_mem_free(bit_ptr);       /* as well, else where did they get it?      */
                                      /* but with some folks, you never know.      */
                                      /* if it does exist, free the memory         */
      op_sim_message("ERROR:DESPREADER2:from SYNC state:tag_and_store_this_pkt().",
                "Failed to reinsert the spreaded bits ptr address into the pkt.
                          Data is Lost.");
  }
  return(its_in_the_bank);
} /* end of: tag_and_store_this_pkt() */
```

```
void
empty_the_list_of_junk_pkts(List *junk_list_ptr, int number_of_junk_pkts)
{
  Packet *junk_pkptr;
  short  *junk_bits_ptr;
  int     ix;

  /* loop through the list, remove the pkt from the list, and destroy it as a lost */
  /* pkt. each removal from the list will make what was the second pkt now the     */
  /* first pkt in the list = OPC_LISTPOS_HEAD. need to also free up the memory     */
  /* each pkt has allocated for the spreaded bits.                                  */

  for(ix=0; ix < number_of_junk_pkts; ++ix)
  {
    junk_pkptr = (Packet *) op_prg_list_remove(junk_list_ptr, OPC_LISTPOS_HEAD);
    if(junk_pkptr != (Packet *) OPC_NIL){
      if(op_pk_nfd_get(junk_pkptr, "SPREADED_DATA", &junk_bits_ptr) ==
                                                         OPC_COMPCODE_SUCCESS)
          clean_up_the_mess(junk_pkptr, junk_bits_ptr);
      else{
          op_sim_message("ERROR:DESPREADER2:empty_the_list_of_junk_pkts()",
                          "FAIL ON GETTING JUNK BIT PTR ADDRESS FROM PKT");
          op_sim_message("UNABLE to Deallocate SPREADED_BITS bytes of memory.","");
      }
    }

    /* else, do nothing. how could a pkptr in the list be null?                  */
    /* else, for some reason, the list of pkt pointers has a pkt address which   */
    /* is a duplicate of one already dealt with. If this is the case, we do not  */
    /* need to process it or release its memory, as that has been done. This     */
    /* should not, forseeably, ever be the case, but it ain't ever supposed to   */
    /* rain on a parade either, so who knows. Calls to op_pk_destroy() with a    */
    /* null pkptr will abort the program, and it should be checked prior.       */
  }
  return;

} /* empty_the_list_of_junk_pkts() */

/*************************************************************************************/
/* Function:   correlate_data_bits(), called from DECODE state, if called at all     */
/* Inputs:  Packet* incoming_pkptr = the spreaded data pkt, need to despread this pkt */
/*         Packet* pkptr2 = a pkt containing our pn_sequence which was recoverd from  */
/*           our synchronization function. If the pkt reached sync, we were able to   */
/*           recover the pn sequence with which is was sent.                          */
/*         int data[] = an array that will hold the data that is recovered           */
/*         int threshold = a simulation parm, user definable, that is the threshold  */
/*           we must meet in our cross correlation in order to accept the data bit.  */
/* Return:     int data_bits = the number of data bits determined to be in the pkt   */
/*                                                                                   */
/* Purpose:    This function is designed to receive a packet of data bits that has   */
/*         been spreaded with some pn sequence. The bits of the respective data pkt  */
/*         are cross correlated with the pn sequence passed in, and compared against */
/*         the threshold passed in. The retrieved data bit is determined to be either */
/*         a 1 (one) or 0 (zero), depending on the correlation value and threshold. If */
/*         the threshold is not met, the value of -1 is set within the array. The    */
```

```c
/*       number of data bits is returned to the calling function for later use.    */
/*                                                                                 */
/* Last Mod:  9/97, r.standfield@computer.org                                      */
/***********************************************************************************/
int
correlate_data_bits(Packet* incoming_pkptr, short* pn_sequence_bit_ptr, short
data[],double threshold)
{
  int     ix, bit_length, code_length, number_of_data_bits;
  short  *int_ptr;
  short  *spreaded_data_bits_ptr;     /* we have a State Var: spreaded_bits_ptr.   */
                                      /* OPNET barks if a functions prototype      */
  short  *bit_ptr;                    /* vars matches a State Var. be careful.     */
  short  *loop_bounds;
  static int      place_holders[3][3] = { { 1,  0, -1},
                                          { 0,  0,  0},
                                          {-1,  0,  1} };

  int   sum = 0;
  double  correlation = 0.0, xcorr = 0.0;

                  /* MAC_BITS, SYNC_BITS, and PN_CODE_LENGTH are defined in mobile.h */
  bit_length          = op_pk_total_size_get(incoming_pkptr);
  number_of_data_bits= (bit_length - SYNC_BITS*SYNC_BITS_PG)/ DATA_BITS_PG - MAC_BITS;
  code_length         = DATA_BITS_PG;

  /* get the pointer to the spreaded data bits. */
  if(op_pk_nfd_get(incoming_pkptr, "SPREADED_DATA", &spreaded_data_bits_ptr) ==
                                                                OPC_COMPCODE_FAILURE)
    op_sim_message("DESPREADER2: correlate_data_bits().",
          "Failed to get memory adress for the information bits. Data is lost");


  /* the pointer obtained is actually pointing to the start of our spreaded data    */
  /* bits. They consist of: 1) spreaded sync bits, 2) spreaded mac bits, and 3) our */
  /* spreaded information bits. All we're really after is the spreaded information */
  /* bits. The sync bits were used to correlate for synchronization in the SYNC     */
  /* state, and apparently passed sync, or we wouldn't be here. we'll effectively   */
  /* 'strip' the sync and mac header bits by skipping past them, then correlating   */
  /* what data bits we have left.                                                   */

  /* set a working ptr to point past the sync and mac bits, starting at the first   */
  /* data bit. Then, loop through the remainder of the spreaded data bits and       */
  /* correlate them individually with our  pn sequence. Our pn sequence is the one  */
  /* which this particular pkt was synchronized on, thus we are reasonably sure it  */
  /* was the same pn sequence used to spread the data. If the correlation meets     */
  /* the user defined threshold (passed in to us), we set the data bit as a 1 or 0  */
  /* accordingly. if the correlation falls below the threshold, we flag the data    */
  /* bit as a -1. The data bits are put in an array that was passed in from the     */
  /* calling function (DECODE State).                                               */

  loop_bounds = pn_sequence_bit_ptr + DATA_BITS_PG;
  bit_ptr = spreaded_data_bits_ptr + SYNC_BITS*SYNC_BITS_PG + MAC_BITS*DATA_BITS_PG;
  for(ix=0; ix < number_of_data_bits; ++ix){
     for(int_ptr = pn_sequence_bit_ptr; int_ptr < loop_bounds; ++bit_ptr, ++int_ptr){
       /* we loop through our pn sequence array a total of data_bits times, as */
       /* each data bit was spreaded with the entire pn sequence.              */
```

```
        sum += place_holders[1 + *bit_ptr][1 + *int_ptr];
      /* sum += (*int_ptr) * (*bit_ptr);  */
    }

    /* set the array value with the respective value, and reset sum = 0*/
    xcorr = (double) sum / code_length;
     data[ix] = (xcorr >= threshold) ? 0 : ( (xcorr <= -threshold) ? 1: -1);
    if(data[ix] == -1)
      update_stats(INFO_BIT_ERROR, ADD_ONE);

    sum = 0;

  } /* end of: for ix < number_of_data_bits */

  /* now that we're done, we no longer need the memory storage for our spreaded  */
  /* data bits. and since the pkt is holding no information and is of no use,   */
  /* what the hell, destroy it too.                                             */

  clean_up_the_mess(incoming_pkptr, spreaded_data_bits_ptr);

  /* we return data_bits. this could have been determined from several different */
  /* ways, and is actually a hold over from earlier evolutions of this coding.   */
  /* It ain't broke, so we aint' fixin' it.                                      */

  return(number_of_data_bits);

} /* end of: correlate_data_bits() */

/***************************************************************************************/
/* Function:   packetize_recovered_data(), called from DECODE state                   */
/* Inputs:     Packet* pktpr = pkt that will hold the data                             */
/*       int data_array[] = array that contains the data. this array was set in the    */
/*         correlate_data_bits() function and should be filled with any and all data   */
/*         bits that were successfully cross correlated with the spreaded data pkt.    */
/*       int number_of_data_bits = the number of data bits that were reportedly rec-   */
/*         overed from the correlate_data_bits() function. It keeps us from running    */
/*         off the end of the array.                                                   */
/* Return:     None                                                                    */
/*                                                                                     */
/* Purpose:    This function is designed to receive an integer array representing data.*/
/*       This is the data that our receiver has 'successfully' recovered after the     */
/*       pkt has passed synchronization, has had the sync bits and the mac header      */
/*       stripped, and the remaining bits of the pkt have been cross correlated        */
/*       with the corresponding PN sequence and the resulting data recovered. As an    */
/*       afterthought, this pkt could have been, or could be constructed at the time   */
/*       the cross correlation is performed.                                           */
/* Last Mod:   9/97, r.standfield@computer.org                                         */
/***************************************************************************************/

Packet*
packetize_recovered_data(short *data_bit_array, int n_data_bits)
{
  short   *data_bits_ptr= (short *)  OPC_NIL;
  Packet  *pkptr      = (Packet *) OPC_NIL;      /* guilty until proven innocent. */

  /* create a pkt that will hold our data */
  pkptr = op_pk_create_fmt("DATA_PKT");
```

```
  if(pkptr != (Packet *) OPC_NIL)                /* don't assume! */
  {
    /* next, allocate enough memory to store the data bits. */
    data_bits_ptr = (short *) op_prg_mem_alloc(n_data_bits*sizeof(short));

    if(data_bits_ptr != (short *) OPC_NIL)                /* any memory available?   */
    {
      /* cool, we have a newly created pkt and a place to store the recovered data.*/
      /* copy over the data bits into this newly allocated memory spot, insert     */
      /* the address into the pkt and we're done. what the hell, print out the pkt */
      /* so the user will think something is going on in here.                     */

      op_prg_mem_copy(data_bit_array, data_bits_ptr, n_data_bits*sizeof(short));

      /* insert the memory address into the pkt.*/
      if(op_pk_nfd_set(pkptr,"DATA", data_bits_ptr, op_prg_mem_copy_create,
                op_prg_mem_free,n_data_bits*sizeof(short)) == OPC_COMPCODE_FAILURE)
      {
        /* if successful, no action required, as the address has been set. If it   */
        /* failed we need to free up the allocated memory, destroy the pkt. and     */
        /* alert the user                                                           */

        clean_up_the_mess(pkptr, data_bits_ptr);
        op_sim_message("ERROR:DESPREADER2:packetize_recovered_data()",
                       "Failed to insert recovered data bits address in pkt.
                                 Data is lost.");
      }
    }
    else    /* else we are having memory problems. bummer. clean up a little.   */
    {
      /* no need for the pkt we created, destroy it. it's !NIL, or we're in    */
      /* the wrong block                                                        */

      clean_up_the_mess(pkptr, data_bits_ptr);
      op_sim_message("ERROR:DESPREADER2:packetize_recovered_data().",
                     "Failed to allocate memory for recovered data bits.
                                              Data is lost.!");
    }
  }
  else /* else we failed to create a pkt of format "DATA_PKT", bail. */
   {
    op_sim_message("ERROR:DESPREADER2:packetize_recovered_data()",
             "Failed to create Packet of format DATA_PKT. Data is lost. so sorry.");
  }

  return(pkptr);   /* it's either still OPC_NIL or has been assigned a good address */

} /* end of: packetize_recovered_data() */
```

```
void update_accumulators(Packet *pkptr)
{
   accumulated_THEO_BER       += op_td_get_dbl(pkptr, OPC_TDA_RA_BER);
   accumulated_ACTUAL_BER     += op_td_get_dbl(pkptr, OPC_TDA_RA_ACTUAL_BER);
   accumulated_NUM_ERRORS     += op_td_get_int(pkptr, OPC_TDA_RA_NUM_ERRORS);
   accumulated_EbNo           += op_td_get_dbl(pkptr, OPC_TDA_RA_MAX_INDEX
                                    + ADDED_TDA_RA_EbNo);
   accumulated_RAYLEIGH_SNR   += op_td_get_dbl(pkptr, OPC_TDA_RA_MAX_INDEX
                                    + ADDED_TDA_RA_RAYLEIGH_SNR);
   accumulated_EbNo_BER       += op_td_get_dbl(pkptr, OPC_TDA_RA_MAX_INDEX
                                    + ADDED_TDA_RA_EBNO_BER);
   return;
} /* end of: update_accumulators() */


void
record_final_stats()
{
   int     total_lost_pkts  = total_pn_collisions + total_sync_failures;

   /* we merely keep track of voice pkt losses. they are not included in the overall pkt
loss prob. */
   double pkt_loss_prob     = (double) total_lost_pkts   / total_slotted_arrivals;
   double voice_pkt_loss_prob= (double) total_voice_drops  / total_slotted_arrivals;

   op_sim_message("DESPREADER2:DESPREAD.","WRITING FINAL ACCUMULATOR STATS");

   /* we assume on this next call, that we are in a single cell, and all xmtrs  */
   /* are matched to our receiver                                                */

   op_stat_scalar_write("Total Number of Users", (double)
   op_topo_object_count(OPC_OBJTYPE_RATX));

   op_stat_scalar_write("Total Incoming Pkts ",(double)  total_incoming_pkts);
   op_stat_scalar_write("Total Good Pkts     ",(double)  total_good_pkts);
   op_stat_scalar_write("Total Voice Drops   ",(double)  total_voice_drops);
   op_stat_scalar_write("Total PN Collisions ",(double)  total_pn_collisions);
   op_stat_scalar_write("Total SYNC Failures ",(double)  total_sync_failures);
   op_stat_scalar_write("Total Info Bit Errs ",(double)  total_info_bit_errors);
   op_stat_scalar_write("Total Runts.        ",(double)  runts);
   op_stat_scalar_write("Minimum ETE Delay.  ",(double)  minimum_ete_delay);
   op_stat_scalar_write("Maximum ETE Delay.  ",(double)  maximum_ete_delay);
   op_stat_scalar_write("Average ETE Delay.  ",(double) (accumulated_ete_delay    /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Theoretical BER     ",(double) (accumulated_THEO_BER     /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Actual BER          ",(double) (accumulated_ACTUAL_BER   /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Total Num Errors    ",(double)  accumulated_NUM_ERRORS );
   op_stat_scalar_write("Avg Num Errors/ pkt ",(double) (accumulated_NUM_ERRORS   /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Average EbNo        ",(double) (accumulated_EbNo        /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Avg Rayleigh SNR    ",(double) (accumulated_RAYLEIGH_SNR /
                                                          total_incoming_pkts));
   op_stat_scalar_write("Average EbNo BER    ",(double) (accumulated_EbNo_BER    /
                                                          total_incoming_pkts));
```

```
  op_stat_scalar_write("Pkt Loss Prob        ",(double)  pkt_loss_prob);
  op_stat_scalar_write("Voice Pkt Loss Prob ",(double)  voice_pkt_loss_prob);
  op_stat_scalar_write("Sim Time (sec)      ",          op_sim_time());

} /* end of: record_final_stats() */


/* the following function is used merely for debugging purposes. it does the obvious. */

void
print_an_array(char *char_string, short int_array[], int n)
{

  int  ix;
  if(*char_string != '\0')              /* if a null string was passed in, skip it,   */
                                        /* else print it                              */
    printf("%s ", char_string);

  for(ix = 0; ix < n; ++ix)       /* printf out the contents, assumed to be short    */
                                  /* ints, like bits                                 */
    printf("%d ", int_array[ix]);

  printf("\n");
} /* end of: print_an_array() */


void clean_up_event_list()
{

  Evhandle  this_event, next_event, temp_event;

  this_event = op_ev_current();
  next_event = op_ev_next_local(this_event);

  while(op_ev_valid(next_event))
  {
    /* check all scheduled events to find any that were scheduled from the SYNC state */
    if((op_ev_type(next_event) == OPC_INTRPT_SELF) && op_ev_code(next_event) ==
                                                                  SYNC_ACHIEVED)
    {
      temp_event = next_event;
      next_event = op_ev_next_local(temp_event);
      if(op_ev_pending(temp_event) == OPC_TRUE)
      {
        /* we have pending events scheduled from the SYNC state. these are events */
        /* that we can cancel, as all pkts that passed sync have been dealt with.  */

        op_ev_cancel(temp_event);
      }
    }
    else
      next_event = op_ev_next_local(next_event);
  }
} /* end of: clean_up_event_list() */
```

```
void
update_stats(int which_stat, float by_how_many)
{
   switch(which_stat)
   {
      case NEW_ARRIVAL:
        total_incoming_pkts   += by_how_many;
        op_stat_write(total_incoming_pkts_stathandle, (double) total_incoming_pkts);
        break;
      case GOOD_PACKET:
        total_good_pkts       += by_how_many;
        op_stat_write(total_good_pkts_stathandle,     (double) total_good_pkts);
        break;
      case PN_COLLISION:
        total_pn_collisions   += by_how_many;
        op_stat_write(current_pn_collisions_stathandle, (double) by_how_many);
        break;
      case VOICE_DROP:
        total_voice_drops     += by_how_many;
        break;
      case SYNC_FAILURES:
        total_sync_failures   += by_how_many;
        break;
      case RUNT_PACKET:
        runts             += by_how_many;
        break;
      case ETE_DELAY:
        accumulated_ete_delay    += by_how_many;
        op_stat_write(ete_delay_stathandle, by_how_many);
        break;
      case INFO_BIT_ERROR:
        total_info_bit_errors += by_how_many;
        break;
      default:
        break;
   }
} /* end of: update_stats() */

void
clean_up_the_mess(Packet *pkptr, short *memory_ptr)
{

   /* check the passed in memory ptr. op_prg_mem_free() will abort the run if passed   */
   /* a null ptr. Too, we sometimes use this function to deallocate memory, or to      */
   /* destory pkts. When doing so, we'll pass in a null ptr for the other parm to keep */
   /* the compiler happy. this null check gives us a little flexibility and hopefully  */
   /* a little quality assurance. if either ptr IS null, just ingore it.               */

   if(memory_ptr != (short *) OPC_NIL)
     op_prg_mem_free(memory_ptr);

   if(pkptr != (Packet *) OPC_NIL)
     op_pk_destroy(pkptr);

} /* end of: clean_up() */
```

INIT State

```
/* let's see who I am? */
my_id = op_id_self ();

/* go get our runtime parameters of the simulation                                         */
/* all of these simulation parameters are user defined for the simulation                  */
/* sync threshold  = the threshold we must meet in order to accomplish synchronization*/
/* data threshold  = analogous to sync threshold, it is the threshold we must meet     */
/*               in order to consider the data information  decodable                 */

/* long_correlation is a user definable Boolean (toggle) simulation parameter. If      */
/* toggled TRUE the following WHILE loop will, if necessary:                          */
/*   for each PN polynomial  (defined in code_matrix[][])                             */
/*   generate a PN Sequence for every  possible initial condition of the shift        */
/*   register  (a phase shifted version of each other)                                 */
/*   correlate the received incoming pkt with each generated PN Sequence               */
/* that can take a bit of simulation time. Without a loss of functionality, long       */
/* long_correlation is defaulted to FALSE. The below WHILE loop will load the shift    */
/* register with the first initial condition = 0x1, and will only check one initial    */
/* condition per PN sequence. this allows dramatic savings in run time simulations,    */
/* without losing the intent. the transmitters shift registers default initial load    */
/* is the same, 0x1.                                                                    */


op_ima_obj_attr_get(my_id, "sync_threshold",    &sync_threshold);
op_ima_obj_attr_get(my_id, "data_threshold",    &data_threshold);
op_ima_obj_attr_get(my_id, "long_correlation", &long_correlation);

/* provide initialization for some of the state variables      */
/* state vars cannot be initialized at declaration. this is    */
/* bad juju, thus we should initialize them here to prevent    */
/* unexpected results                                          */

total_incoming_pkts     = 0;
total_good_pkts         = 0;
total_voice_drops       = 0;
total_pn_collisions     = 0;
total_sync_failures     = 0;
total_info_bit_errors   = 0;

safe_keeping            = OPC_FALSE;

accumulated_ete_delay   = 0.0;
accumulated_THEO_BER    = 0.0;
accumulated_ACTUAL_BER  = 0.0;
accumulated_NUM_ERRORS  = 0.0;
accumulated_EbNo        = 0.0;
accumulated_RAYLEIGH_SNR= 0.0;
accumulated_EbNo_BER    = 0.0;

/* based upon REGISTER_BIT_LENGTH defined in mobile.h, determine */
/* how long our pn sequence should be.                           */

max_number_of_shift_register_states    = pow(2,REGISTER_BIT_LENGTH) - 1;
```

```
max_number_of_correlations_to_compute =
    long_correlation == OPC_FALSE ? 1: max_number_of_shift_register_states;
                                                    /* FALSE is most likely */


/* register our statistics handle, allowing our probe file to sample the output   */
total_incoming_pkts_stathandle    = op_stat_reg("Total Incoming Pkts",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
total_good_pkts_stathandle        = op_stat_reg("Total Good Pkts",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
ete_delay_stathandle              = op_stat_reg("ETE Delay",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);
current_pn_collisions_stathandle  = op_stat_reg("Current PN Collisions",
                        OPC_STAT_INDEX_NONE, OPC_STAT_LOCAL);


/* This model is intended to model a CDMA SS system. Thus, we want to allow          */
/* simultaneous multiple access within this receiver. In our SYNC state, we          */
/* do several things, which we will see, but a note is worth explaining now.         */
/* We'll have to correlate each packet that is received to determine if we can       */
/* decode the pkt. Errors induced in the error.ps.c pipeline stage could render      */
/* the pkt as junk of course. Assume the case of simultaneous reception of two pkts. */
/* Assuming they both were correlated fine and were determined as 'good' pkts, we    */
/* need to also determine if they were sent using the same or different PN Codes.    */
/* If the same, then we have a 'true collision' and the pkts get destroyed. If       */
/* different codes, then we're ok. As pkts will assumingly arrive at the receiver    */
/* at the same time (equidistant xmtrs, slotted MAC), we need to maintain all of     */
/* the pkts until we have processed all that were simultaneously received. Once all  */
/* have been processed, we check for duplicate PN codes and deal with those pkts then. */

/* we accomplish this by placing the pkts in a linked list. the below array is the   */
/* array of pointers pointing to the linked lists. The indices of the array will     */
/* match the index of the pn sequence which the received pkt used, based on the      */
/* polynomial that was used in the transmitter. The polynomials are defined in the   */
/* header block of this process, in code_matrix[][].                                  */

/* For example: Assume two pkts were received simultaneously, and both used the first */
/* polynomial, R(3,7) to determine their pn sequence. The end result is               */
/* array_of_ptrs_to_packet_list[0] will  point to a linked list, and the elements of  */
/* this list will be the pkt pointers of both received pkts, size of list = 2.        */

/* once all the simultaneous received pkts have been gone through, we merely go      */
/* through this array and find the size of each list. If the list has N > 1 elements, */
/* this indicates N pkts were simultaneously received  that had the same PN code.     */
/* We destroy them and adjust our counters, which we will see.                        */

/* for now, create the lists */
for(ix=0; ix < NUMBER_PN_CODES; ++ix)
  array_of_ptrs_to_packet_list[ix] = op_prg_list_create();

/* set the initial condition of the shift register.  */
set_register_to_first_state(shift_register, REGISTER_BIT_LENGTH);
```

```
for(ix = 0; ix < NUMBER_PN_CODES; ++ix)
{
  set_register_to_first_state(shift_register, REGISTER_BIT_LENGTH);
  for(jx = 0; jx < NUMBER_M_SEQUENCES; ++jx)
  {
    generate_pn_sequence(matrix_of_pn_sequences[ix][jx].bit_array, shift_register, ix);
    get_next_register_state(shift_register, REGISTER_BIT_LENGTH);

  }
}
```

SYNC State

```
/* there is only one place for pkts to come from, as this receiver has only one   */
/* channel. the only stream intrpt to expect is from the uplink. Get the packet   */

received_pkptr = op_pk_get (op_intrpt_strm ());
if(received_pkptr != (Packet *) OPC_NIL)
{
  /* we need to get the pointer, from the pkt, that is pointing to the memory     */
  /* location of our bits. Recall that our packet only contains one field,        */
  /* SPREADED_DATA, and this field is defined as a strucutre. In actuality, it is*/
  /* a pointer to the memory location of all of our spreaded bits. Only the       */
  /* pointer traversed the pipeline, and the bits remained 'safely' inside of     */
  /* memory. The total size of the pkt was set in the PN_SPREADER process, which */
  /* allowed the pipeline stages to perform their necessary calculations. Now, we*/
  /* need to extract this pointer and manipulate the bits. Specifically,          */
  /* correlate the sync bits. Recall as well, that accessing the structure field */
  /* of this pkt will  encapsulate the bits from the pkt. We'll need to  'reset'*/
  /* this pointer back into the pkt before we leave this state, as the DECODE     */
  /* state will need it to find and decode the information bits.                  */

  spreaded_bits_ptr = process_the_new_guy(received_pkptr);
  if(spreaded_bits_ptr != (short *) OPC_NIL)
  {
    do
    {
      /* loop through, as necessary, all the PN codes and their variants */
      for(ix = 0; (ix < NUMBER_PN_CODES) && (!achieved_sync); ++ix){
        for(jx=0; jx < max_number_of_correlations_to_compute; ++jx){
          /* correlate the two packets    */
         sync_correlation =
          correlate_sync_bits(spreaded_bits_ptr,matrix_of_pn_sequences[ix][jx].bit_array
          );

          /* if magnitude(sync correlation) >= magnitude(our user defined threshold) */
          /* we're golden.                                                            */
          if(fabs(sync_correlation) >= sync_threshold)
          {
            achieved_sync = OPC_TRUE;
            array_of_ptrs_to_pn_sequences[ix] =
                                matrix_of_pn_sequences[ix][jx].bit_array;

            /* need to maintain this pkt as discussed in the INIT state.         */
            /* tag_and_store_this_pkt()will attempt to reinsert the              */
            /* the spreaded_bits_ptr into the pkt, and if successful, will       */
```

```
            /* attempt to place this pkt in the list being pointed to by the      */
            /* pointer element, array_of_ptrs_to_packet_list[ix]. This 'ix'       */
            /* element is pointing to a list which consists of a pointers to      */
            /* pn sequences which match the pn sequence that the respective       */
            /* pkt was able to synchronize on, such as this one. If sync has      */
            /* been achieved, then the pn sequence with which it was synchronized */
            /* on is found in the 3D matrix,                                      */
            /* matrix_of_pn_sequences[ix][jx].bit_array. bit_array is a defined   */
            /* structure of short[].ix was assigned to point to  this particular  */
            /* sequence in the above assignment. we now want to store this pointer*/
            /* for safekeeping.                                                   */

            safe_keeping = tag_and_store_this_pkt(received_pkptr,
                           spreaded_bits_ptr,array_of_ptrs_to_packet_list[ix]);

            /* if we somehow managed to sync on a pkt, but couldn't restore       */
            /* the data bits pointer into the pkt, we've lost our data. we'll     */
            /* chock this one up to Runts.                                        */

            if(!safe_keeping)
              update_stats(RUNT_PACKET, ADD_ONE);

            /* we've achieved sync on this pkt, so let's bail out of this loop */
            break;
          } /* end of: if(fabs(sync_correlation) >= sync_threshold) */
        } /* end of: for jx < upper_bound */
      } /* end of: for ix < NUMBER_PN_CODES) && (!achieved_sync)  */

      /* we've now been through all the polynomials defined in code_matrix[][]. If */
      /* we haven't been able to sync yet, we're hosed, as we have checked all     */
      /* possible pn sequences and all phases (dependent on long_correlation)      */

      all_PN_codes_checked = OPC_TRUE;

    } while( ! (achieved_sync || all_PN_codes_checked));

    if(achieved_sync)
      op_intrpt_schedule_self(op_sim_time() + TAU, SYNC_ACHIEVED);
                                      /* need a time delay, see INIT comments */

    else    /* this pkt is hopeless junked bits. we'll destroy it and */
            /* release the memory                                     */
    {
      /* increment our counters, deallocate used memory, and destroy the trashed pkt.*/
      update_stats(SYNC_FAILURES, ADD_ONE);
      look_at_stats_of_pkt(received_pkptr);
      clean_up_the_mess(received_pkptr, spreaded_bits_ptr);
    } /* end else */
  }  /* end of: if(spreaded_bits_ptr != (short *) OPC_NIL)  */
}  /* end of: if(received_pkptr != (Packet *) OPC_NIL)       */

else /* the pkt is somehow a NIL pkt, can't do anything with it. */
  op_sim_message("DESPREADING PROCESS: SYNC STATE: PKT FROM STREAM IS NIL?", "No Clue.");
```

DECODE State

```
/* we're here only because we were able to achieve synchronization on the              */
/* received spreaded packet  in the SYNC state. If sync was achieved, the received     */
/* pkt was placed inside a list (acutally an  an array of list pointers). As the       */
/* SYNC state will be invoked for each received pkt at time = END_RX, we need to       */
/* to maintain all pkts that passed synchronization until all received pkts  have      */
/* have been processed. Once OPNET has processed all the events in the event list      */
/* (all the STRM_INTRPT's), we can then safely  /* move to this DECODE state and       */
/* attempt to decode the data of the received pkt. We now have an array of list        */
/* pointers, each pointing to a list of pkts  which were synchronized with the same    */
/* PN code sequence (if any). If there are more than one pkt in each list, we have     */
/* a collision caused by multiple PN code sequences.                                   */

/* now that we are here, we'll need to correlate the mac and data pkt and extract     */
/**/
/* the information.We know from the correlation process of the synchronization bits */
/* what particular characteristic polynomial and what initial register load was       */
/* used by the transmitter. That info has been  maintained in our state variables.   */
/* From that info, we have also already discovered our PN sequence that was used      */
/* to spread the information on the transmitter end. At this point, we have a pkt     */
/* of spreaded information which has been stripped of the synchronization bits.       */

/* Recall our array_of_ptrs_to_packet_list[] was declared to hold pointers to a       */
/* list of pkt pointers. We need to loop through the array, check each list pointed */
/* to by the array element. If there are more than one elements in each respective   */
/* list, this indicates we had packets with the same PN sequence. Deal with it.       */

for(ix=0; ix < NUMBER_PN_CODES; ix++)
{
  /* how many pkts are in the list pointed to by array_of_ptrs_......_list[ix]    */
  number_of_packets_in_list = op_prg_list_size(array_of_ptrs_to_packet_list[ix]);

  /* if there are N > 1 items in the list, we have colliding sequences = colliding pkts.
  */
  /* any colliding packets are considered junk pkts, as we cannot determine from      */
  /* which user they originated from. This is the cool aspect of CDMA and this        */
  /* slotted ALOHA business, in that we can have multiple users accessing the         */
  /* network and only those with duplicate PN codes are considered truly collisions.*/

  if(number_of_packets_in_list > 1)
  {
    update_stats(PN_COLLISION,number_of_packets_in_list);
    empty_the_list_of_junk_pkts(array_of_ptrs_to_packet_list[ix],
                                number_of_packets_in_list);
  }

    /* else if there is only one element in the list, ok. we received a pkt with      */
    /* the corresponding PN sequence polynomial. Get this pkt from the list,          */
    /* then:                                                                          */
    /*   strip the synchronization bits from the pkt.                                 */
    /*   strip the MAC header bits from the pkt                                       */
    /*   correlate what's left = only data and attempt to recover the data            */
    /*     and don't forget to destroy the original pkt remaining in the list.       */
```

```
else if(number_of_packets_in_list > 0)
{
  /* if only one pkt in the list, it's a successful reception. update our stats   */

  update_stats(GOOD_PACKET, ADD_ONE);

  /* We're in this branch because there was only one pkt in the linked list,      */
  /* which indicates there were  no PN Collisions. The pkt currently has a sync   */
  /* header and a mac header prefacing the information data bits, and all of this*/
  /* has been spreaded by our PN sequence in the transmitter. We recovered this   */
  /* pn sequence in the SYNC state (or we wouldn't be here). What is left to do   */
  /* is to strip the sync bits, strip the mac header bits, then despread          */
  /* (correlate) the data bits and attempt to recover as many as we can. The      */
  /* error.ps.c pipeline stage was our error injector. Based on the determined    */
  /* BER, this error pipeline stage may very well have ruined some of our bits.   */
  /* Recall that our pkt only contains a pointer to the spreaded bits in memory. */
  /* Rather than actually stripping the sync and mac header bits from the 'pkt'   */
  /* we'll pass this pkt to our correlate_data_bits() function which will:        */
  /*   extract the spreaded_bits_ptr from the pkt, skip  past the sync and mac    */
  /*   header bits, then correlate the bits from that point on.                   */
  /* The correlated data values will be returned in data_array[]. The memory      */
  /* allocated and pointed to by our memory ptr *spreaded_bits_ptr will be        */
  /* deallocated within correlate_data_bits()                                     */


  /* first, get (effectively removing) the pkt from the linked list.we can        */
  /* either access or remove an element in the list. Accessing allows             */
  /* non-destructive querying values of the pkt, whereas removal will actually    */
  /* remove the item from the list. wow, what a conept huh.                       */

  /* there should only be one packet in the list pointed to by                    */
  /* array_of_ptrs_to_packet_list[ix] at this point. get a pointer to it.         */

  spreaded_data_pkptr = (Packet*) op_prg_list_remove(array_of_ptrs_to_packet_list[ix],
                                                       OPC_LISTPOS_HEAD);

  if(spreaded_data_pkptr != (Packet *) OPC_NIL)    /* never assume! */
  {
    number_of_data_bits =
         correlate_data_bits(spreaded_data_pkptr, array_of_ptrs_to_pn_sequences[ix],
                                                      data_array, data_threshold);

    /* we should now have an array containing the recovered transmitted data      */
    /* bits (assuming the signal was legit). we'll 'repacketize' these data       */
    /**/
    /* data bits and send them on their merry way. In this case, their merry      */
    /* way is considered to be the next protocol layer, like maybe the MAC        */
    /* layer. Our MAC layer consists of a SINK process which will just destroy    */
    /* the pkt. We could do it here as well, but our intent is to complete this   */
    /* entire path from tx to rx. With this approach, we can easily use a module */
    /* other than SINK for our next process, and minimal recoding is required.   */
    /* This keeps this implementation somewhat cohesive.                          */

    /* number_of_data_bits tells us how many data bits are in the pkt. */
    recovered_data_pkptr = packetize_recovered_data(data_array, number_of_data_bits);
```

```
        if(recovered_data_pkptr != (Packet *) OPC_NIL)
          op_pk_send(recovered_data_pkptr,TO_HIGHER_LAYER);
        else
          op_sim_message("ERROR:DESPREADER:DECODE STATE:",
                    "Unable to recover data bits. Your data from this packet is lost.");
      }
      else;  /* else the SYNC state inserted a null pkptr into our list, the jerk    */
             /* who knows why.  now action required, as there is nothing to destroy  */


    }  /* end of: else if(number_of_packets_in_list > 0) */
  }  /* end of: ix < NUMBER_PN_CODES  */

/* we have effectively recovered any data pkts that were received collision free.      */
/* this includes packets that suffered MAI from other arriving pkts w/different PN     */
/* codes likewise, we have destroyed any pkts that were collisions in the sense        */
/* that they had the same PN sequence. Thisprocess was invoked due to a self           */
/* interrupt scheduled  in the SYNC State.                                             */
/* Thus, there are as many interrupts waiting to invoke this process                   */
/* as there were multiple pkts received. We're probably best off by going through the */
/* event list and removing the remaining events scheduled for this process. All the    */
/* necessary work to recover all non-colliding packets and interfered packets has been*/
/* been done in this first invocation. As well, all colliding packets have been        */
/* destroyed, counters adjusted, and the linked list(s) pointed to by                  */
/* array_of_ptrs_to_packet_list[] has been gone through, destoying any packets that    */
/* were in this list, as they have all been dealt with accordingly. Good housekeeping */
/* suggests we clean up our mess. thus, we'll loop through the event list checking for*/
/* any remaining INTRPT_SELF's. If they are of code SYNC_ACHIEVED, then they were      */
/* scheduled in the SYNC state. There is at least one other INTRPT_SELF currently      */
/* scheduled, an END_SIM interrupt, so don't kill it. The only apparent danger, at     */
/* this stage, is that there could have been arriving packets in the next slot time    */
/* that have been through the SYNC state, and thus have scheduled a INTRPT_SELF for    */
/* themselves.                                                                         */

/* However, the interrupt scheduled in the SYNC state is with a nominal time delay,    */
/* call it tau, after completing the SYNC state. as long as tau < guard time,          */
/* we should be safe. As a complete check, we can check the approximate scheduled time*/
/* of the events, and cancel only those that are scheduled for about this time, which */
/* would be those remaining interrupts scheduled in this slot time.                    */

/* now I'm thoroughly confused. maybe the code will be clear.                          */

clean_up_event_list();
```

LIST OF REFERENCES

1. FORWARD . . . FROM THE SEA, Dept. of the Navy, Washington D.C., 1994.

2. Duff, D. A., *Wireless Applications for Marine Air Ground Task Forces*, Master's Thesis, Naval Postgraduate School, Monterey, California, June 1996.

3. Cummiskey, J. C., *Internetworking: The Interoperability of Commercial Mobile Computers with the USMC Digital Automated Communications Terminal (DACT)*, Master's Thesis, Naval Postgraduate School, Monterey, California, September, 1996.

4. Gingras, D. F., *Littoral Forces Communications*, SPAWAR Systems Center, San Diego, California, Brief to Marine Corps Tactical Systems Support Activity (MCTSSA), Camp Pendleton, California, March 1997.

5. Schwartz, M., *Broadband Integrated Networks*, Prentice-Hall, 1996.

6. Uziel, A. and Tummala, M., *Modeling of Low Data Rate Services for Mobile ATM*, Proc. Personal Indoor and Mobile Radio Comm. Conf. '97, pp. 194-198, 1997.

7. Brady, P. T., *A Model for Generating On-Off Speech Patterns in Two-Way Conversations*, Bell System Technical Journal, pp. 2445-2472, September, 1969.

8. Rappaport, T. S., *Wireless Communications*, Prentice-Hall, 1996.

9. Stallings, W., *Data and Computer Communications*, Fifth Edition, Prentice-Hall, 1997.

10. Peterson, R. L., Ziemer, R. E., and Borth, D. E., *Introduction to Spread Spectrum Communications*, Prentice-Hall, 1995.

11. Robertson, C., *Personal Communications*, Naval Postgraduate School, Monterey, California, August, 1997.

12. Sklar, B., *Digital Communications, Fundamentals and Applications*, Prentice-Hall, 1988.

13. Lee, W. C. Y., *Overview of Cellular CDMA*, IEEE Trans. Veh. Tech., vol. 40, no. 2, pp. 291-302, May, 1991.

14. Proakis, J. G., *Digital Communications*, Third Edition, McGraw-Hill, 1995.

15. Pursley, M. B., *Performance Evaluation for Phase-Coded Spread Spectrum Multiple-Access Communication---Part I: System Analysis*, IEEE Trans. Comm., vol. COM-25, no. 8, pp. 795-799, August 1977.

16. Brand, A. E. and Aghvami, A. H., *Performance of a Joint CDMA/PRMA Protocol for Mixed Voice/Data Transmission for Third Generation Mobile Communication*, IEEE J. Select. Areas Comm., vol. 14, no. 9, pp. 1698-1707, December, 1996.

17. OPNET, *Modeling*, vol.'s 1-2, MIL 3, Inc. 3400 International Drive NW, Washington D.C., 20008, 1996.

18. Larson, H. J. and Shubert, B. O., *Probabilistic Models in Engineering Sciences*, John Wiley, 1979.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center.. 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library.. 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Director, Training and Education 1
   MCCDC, Code C46
   1019 Elliot Rd.
   Quantico, Virginia 22134-5027

4. Director, Marine Corps Research Center 1
   MCCDC, Code C40RC
   2040 Broadway Street
   Quantico, Virginia 22134-5107

5. Director, Studies and Analysis Division... 1
   MCCDC, Code C45
   300 Russell Road
   Quantico, Virginia 22134-5130

6. Dr. Don Gingras.. 1
   SPAWAR Systems Center San Diego, Code D8805
   Communication and Information Systems Department
   San Diego, California 92152-5001

7. Professor Murali Tummala.. 3
   Department of Electrical and Computer Engineering, Code EC/Tu
   Naval Postgraduate School
   Monterey, California 93943-5000

8. Professor Tri Ha... 1
   Department of Electrical and Computer Engineering, Code EC/Ha
   Naval Postgraduate School
   Monterey, California 93943-5000

9, Major Amir Uziel 1
   Department of Electrical and Computer Engineering, Code EC/Tu
   Naval Postgraduate School
   Monterey, California 93943-5000

10. Captain Roger Standfield.. 2
PO Box 2268
Oceanside, California 92051-2268